WELCOME!

(download slides and .py files from the class site to follow along)

6.100L Lecture 1

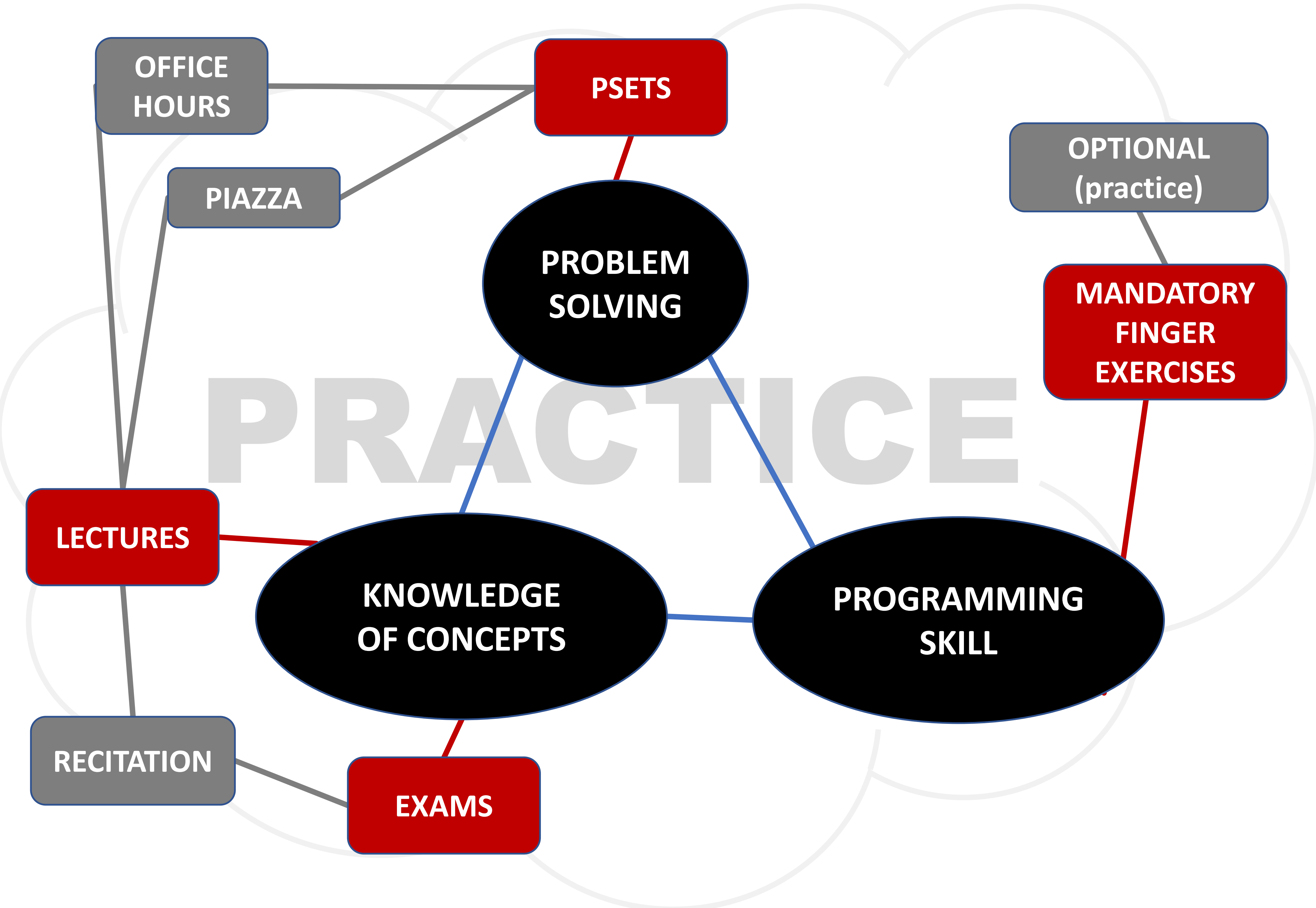
Ana Bell

TODAY

- Course info
- What is computation
- Python basics
  - Mathematical operations
  - Python variables and types
- NOTE: **slides and code files up before each lecture**
  - Highly encourage you to download them before class
  - Take notes and run code files when I do
  - Do the in-class “You try it” breaks
  - Class will not be recorded
  - Class will be live-Zoomed for those sick/quarantine

WHY COME TO CLASS?

- You get out of this course what you put into it
- Lectures
  - **Intuition** for concept
  - **Teach** you the concept
  - **Ask** me questions!
  - **Examples** of concept
  - Opportunity to
    practice practice practice
  - Repeat


OFFICE HOURS
PSETS
OPTIONAL (practice)
PIAZZA
PROBLEM SOLVING
MANDATORY FINGER EXERCISES
PRACTICE
LECTURES
KNOWLEDGE OF CONCEPTS
PROGRAMMING SKILL
RECITATION
EXAMS

TOPICS

- Solving problems using **computation**
- Python **programming language**
- Organizing **modular programs**
- Some simple but important **algorithms**
- Algorithmic **complexity**

LET’S GOOOOO!

TYPES of KNOWLEDGE

- **Declarative knowledge** is **statements of fact**
- **Imperative knowledge** is a **recipe** or “how-to”

- Programming is about writing recipes to generate facts

NUMERICAL EXAMPLE

- Square root of a number `x` is `y` such that `y*y = x`
- Start with a **guess**, `g`
  1) If `g*g` is **close enough** to `x`, stop and say `g` is the answer
  2) Otherwise make a **new guess** by averaging `g` and `x/g`
  3) Using the new guess, **repeat** process until close enough
- Let's try it for x = 16 and an initial guess of 3

g	g*g	x/g	(g+x/g)/2
3	9	16/3	4.17

NUMERICAL EXAMPLE

- Square root of a number `x` is `y` such that `y*y = x`
- Start with a **guess**, `g`
  1) If `g*g` is **close enough** to `x`, stop and say `g` is the answer
  2) Otherwise make a **new guess** by averaging `g` and `x/g`
  3) Using the new guess, **repeat** process until close enough
- Let's try it for x = 16 and an initial guess of 3

g	g*g	x/g	(g+x/g)/2
3	9	16/3	4.17
4.17	17.36	3.837	4.0035

NUMERICAL EXAMPLE

- Square root of a number `x` is `y` such that `y*y = x`
- Start with a **guess**, `g`
  1) If `g*g` is **close enough** to `x`, stop and say `g` is the answer
  2) Otherwise make a **new guess** by averaging `g` and `x/g`
  3) Using the new guess, **repeat** process until close enough
- Let's try it for x = 16 and an initial guess of 3

g	g*g	x/g	(g+x/g)/2
3	9	16/3	4.17
4.17	17.36	3.837	4.0035
4.0035	16.0277	3.997	4.000002

WE HAVE an ALGORITHM

1) Sequence of simple **steps**

2) **Flow of control** process that specifies when each step is executed

3) A means of determining **when to stop**

ALGORITHMS are RECIPES / RECIPES are ALGORITHMS

- Bake cake from a box
  - 1) Mix dry ingredients
  - 2) Add eggs and milk
  - 3) Pour mixture in a pan
  - 4) Bake at 350F for 5 minutes
  - 5) Stick a toothpick in the cake
    - 6a) If toothpick does not come out clean, repeat step 4 and 5
    - 6b) Otherwise, take pan out of the oven
  - 7) Eat

COMPUTERS are MACHINES that EXECUTE ALGORITHMS

- Two things computers do:
  - Performs simple **operations**
    100s of billions per second!
  - **Remembers** results

    100s of gigabytes of storage!
- What kinds of calculations?
  - **Built-in** to the machine, e.g., +
  - Ones that **you define** as the programmer
- The BIG IDEA here?

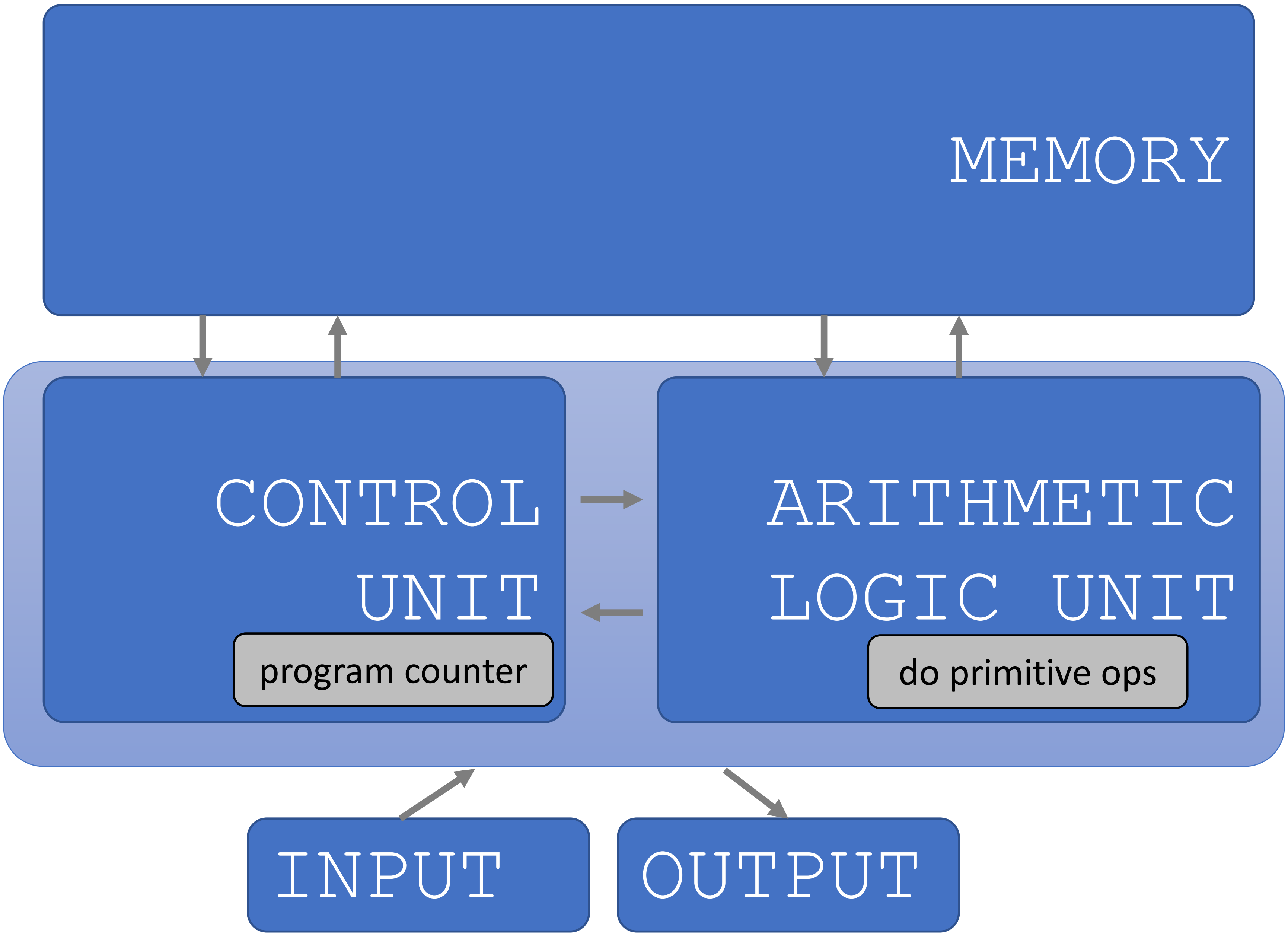
A COMPUTER WILL ONLY DO WHAT YOU TELL IT TO DO

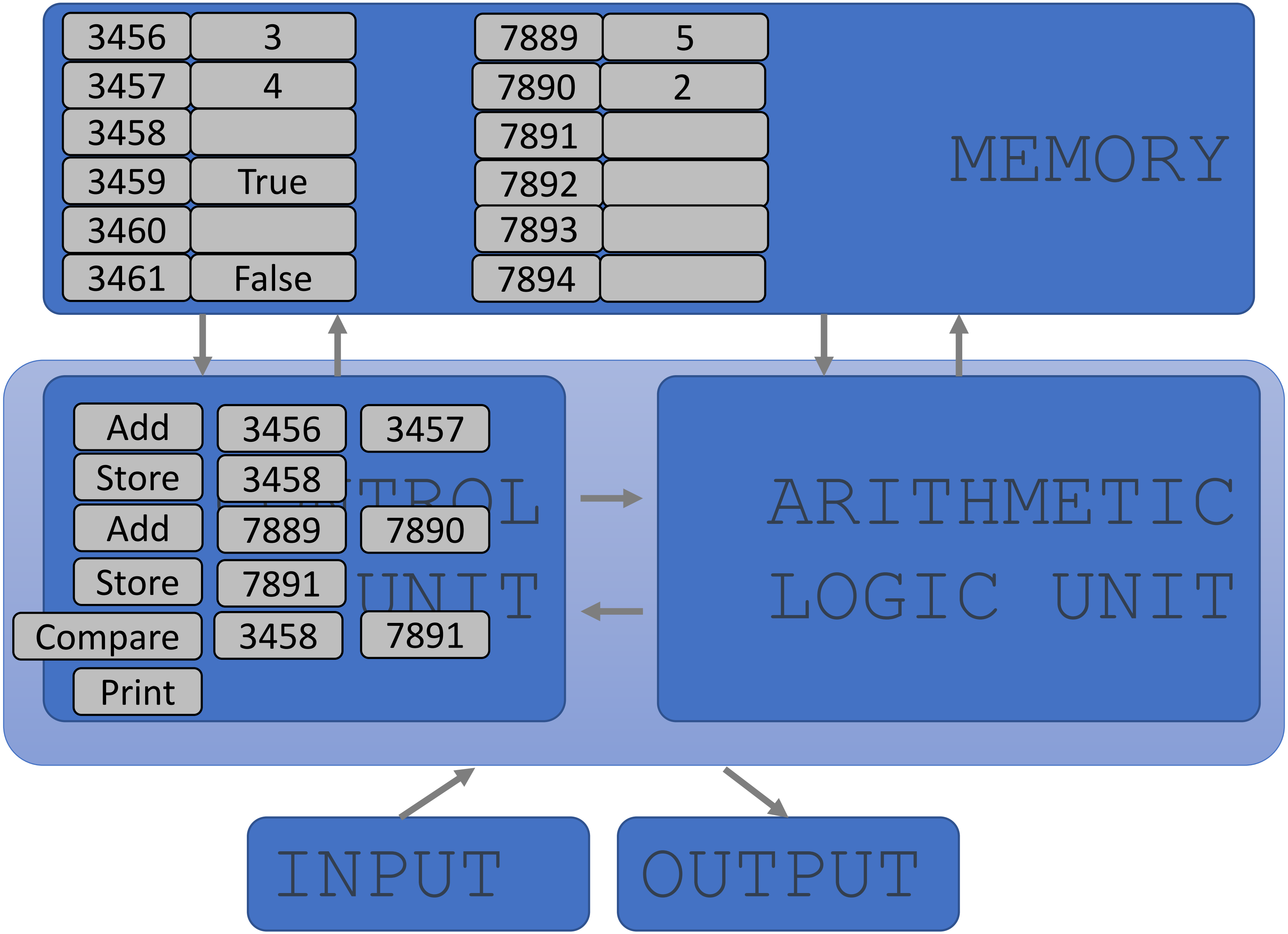
COMPUTERS are MACHINES that EXECUTE ALGORITHMS

- **Fixed program** computer
  - Fixed set of algorithms
  - What we had until 1940's
- **Stored program** computer
  - Machine stores and executes instructions
- **Key insight**: Programs are no different from other kinds of data

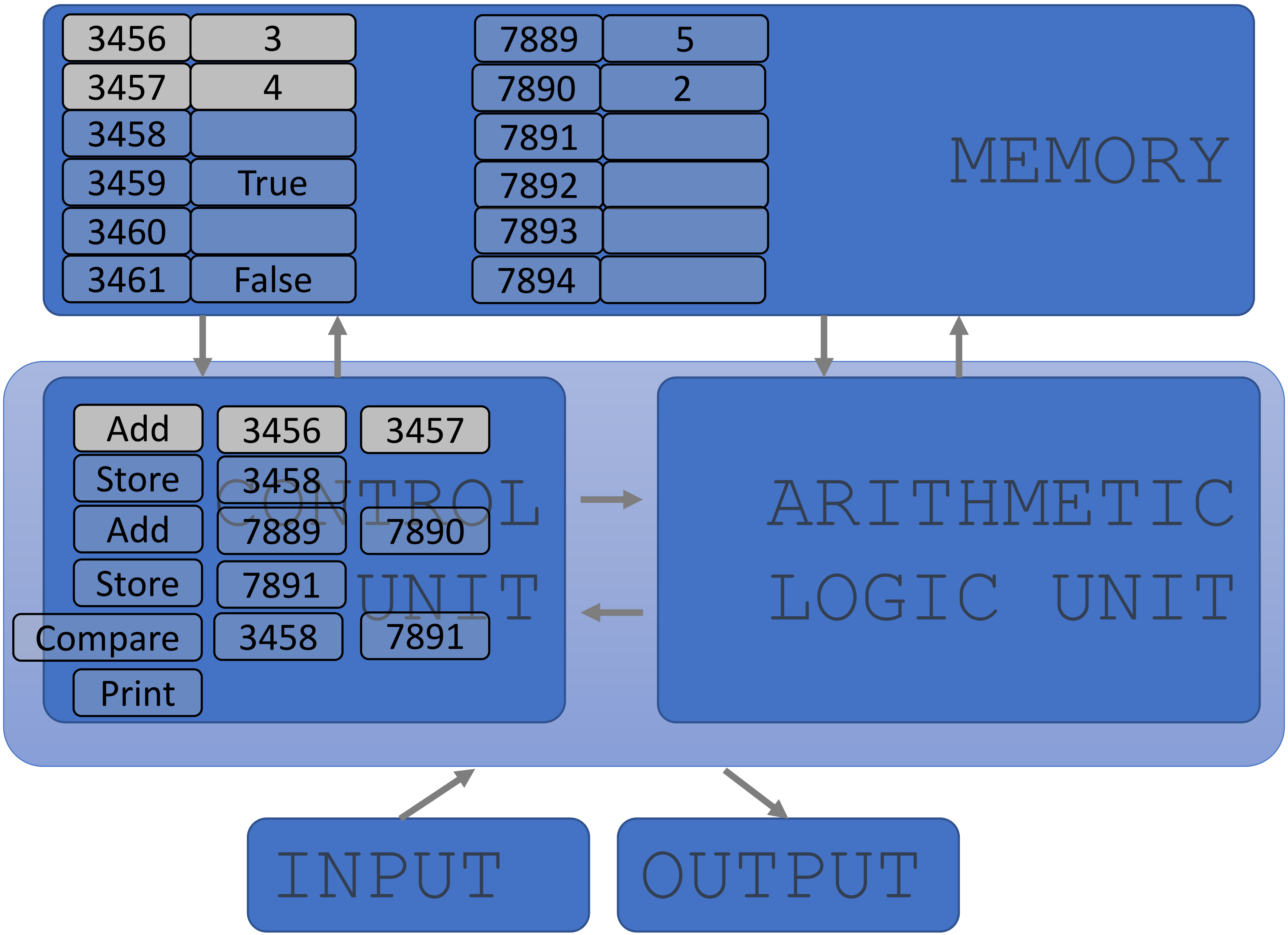
STORED PROGRAM COMPUTER

- Sequence of **instructions stored** inside computer
  - Built from predefined set of primitive instructions
  1) Arithmetic and logical
  2) Simple tests
  3) Moving data
- Special program (interpreter) **executes each instruction in order**
  - Use tests to change flow of control through sequence
  - Stops when it runs out of instructions or executes a halt instruction

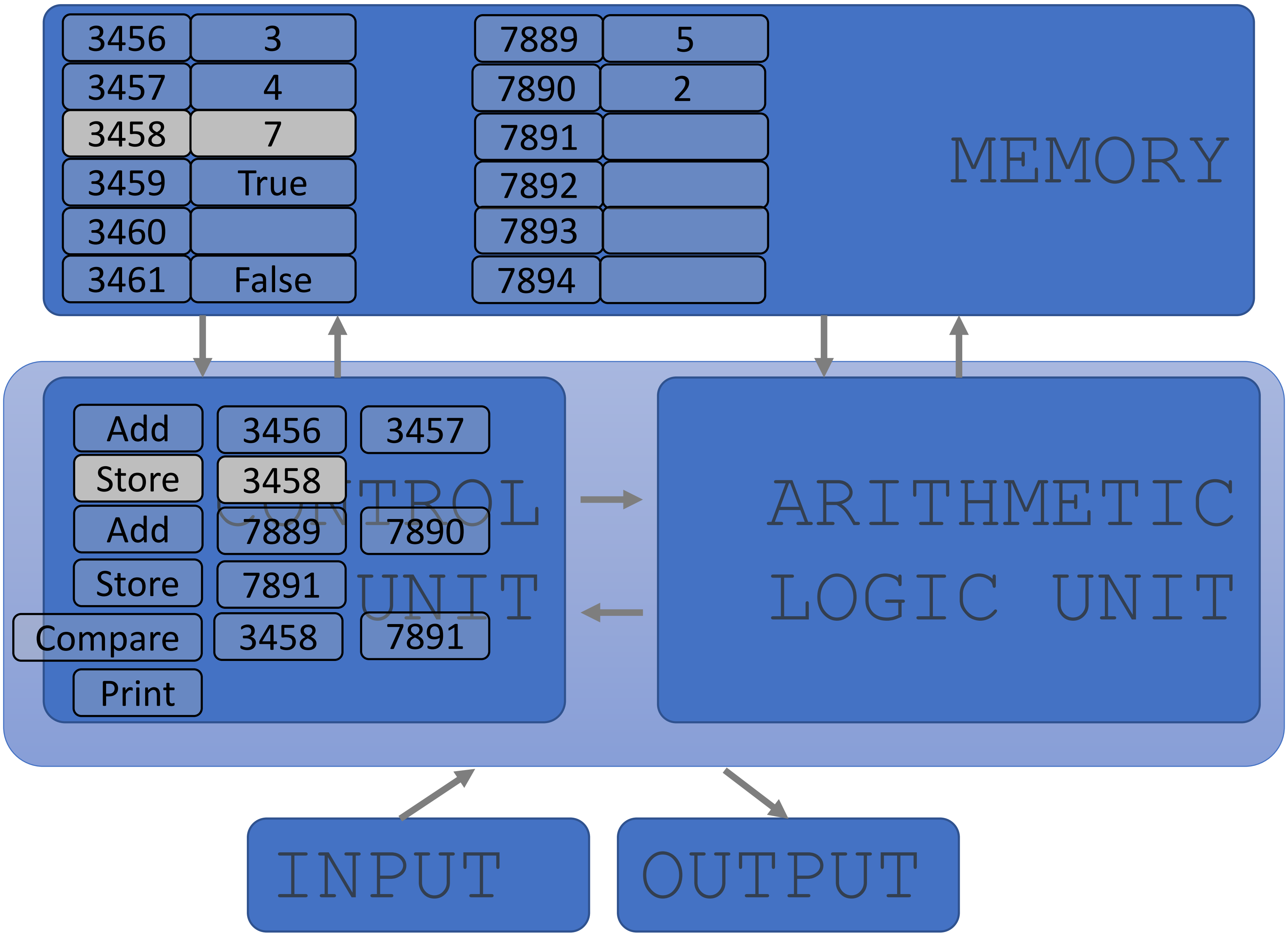

MEMORY
CONTROL UNIT
program counter
ARITHMETIC LOGIC UNIT
do primitive ops
INPUT
OUTPUT

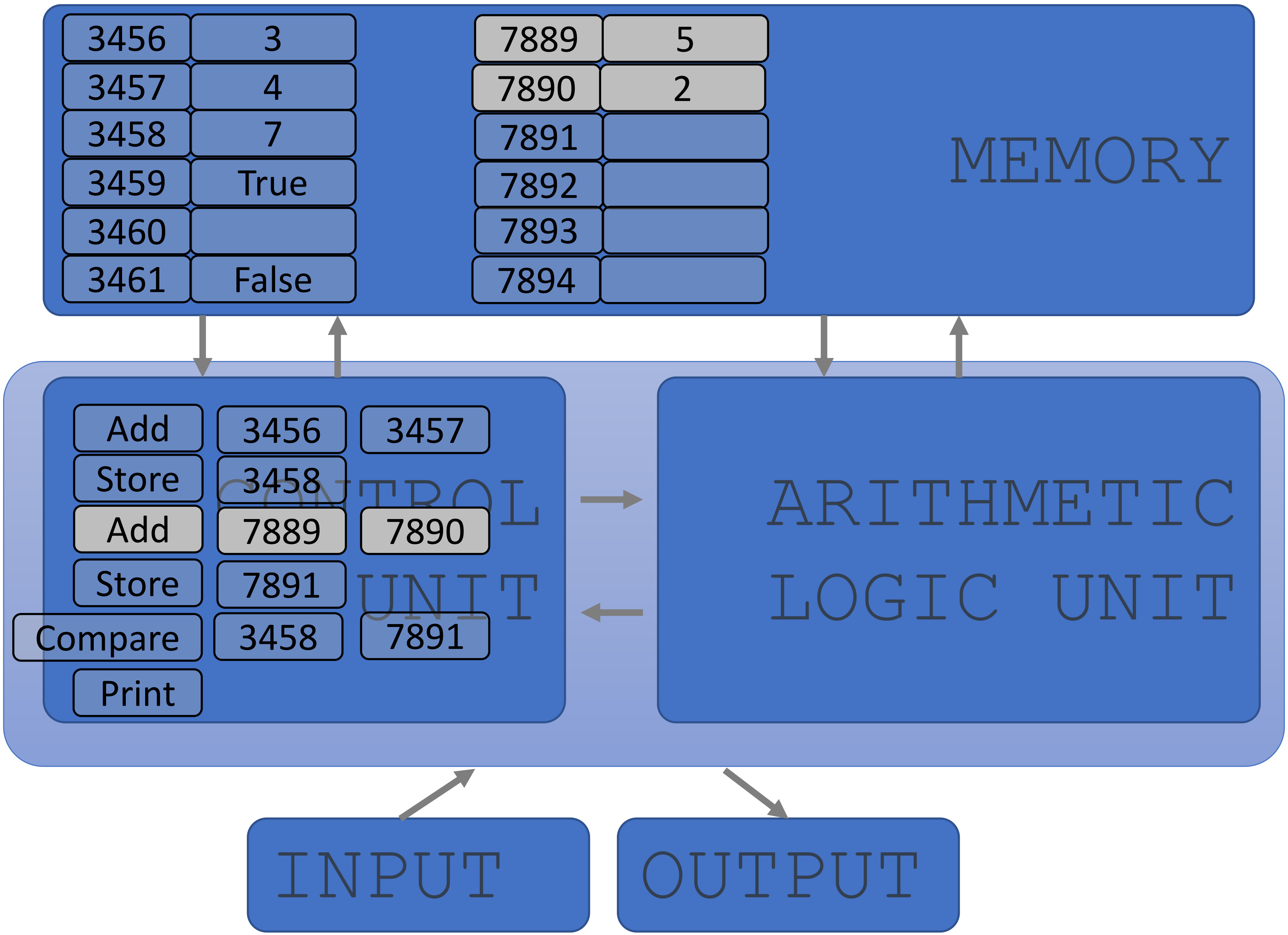


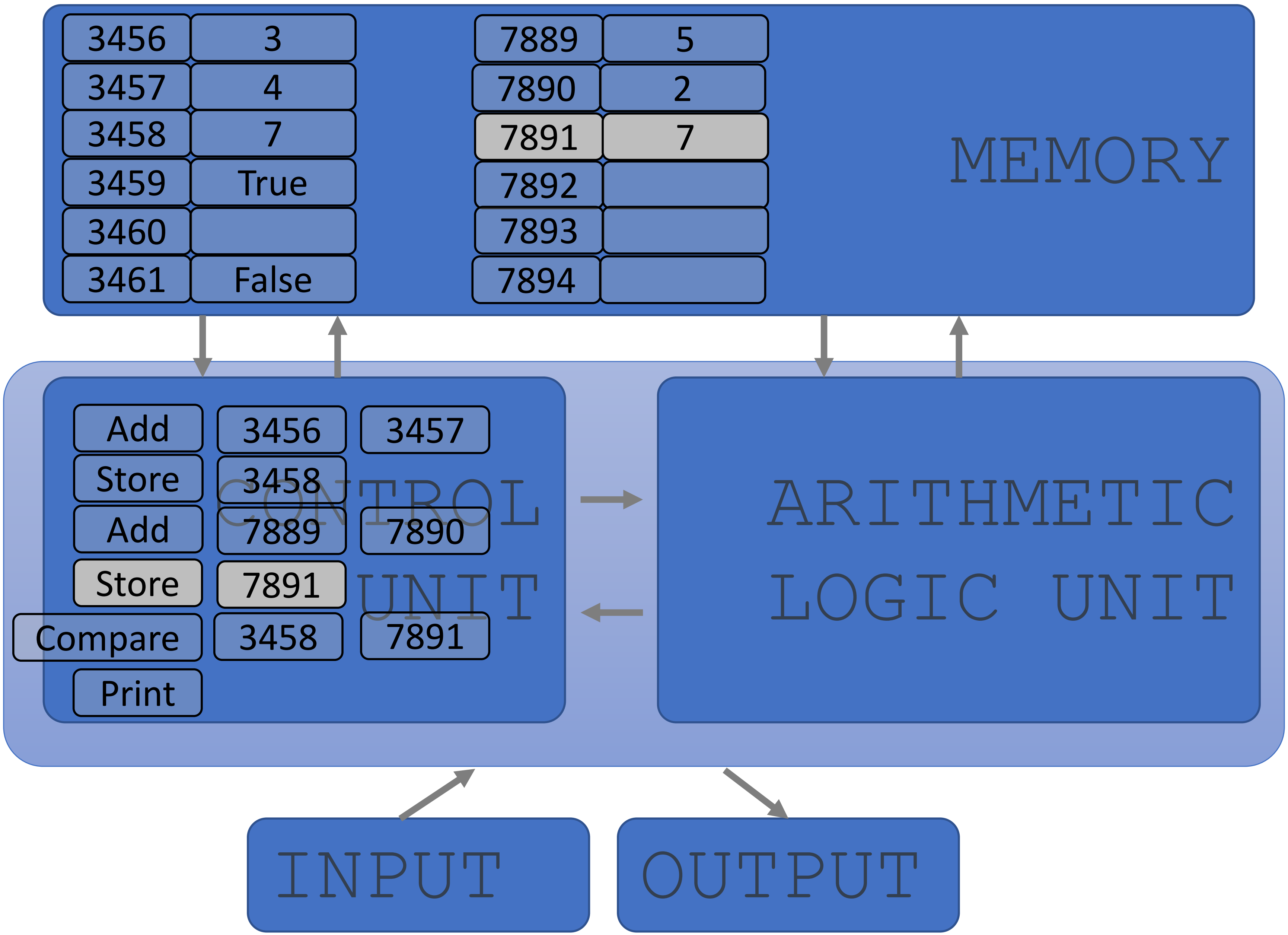
3456
3
3457
4
3458
3459
True
3460
3461
False
7889
5
7890
2
7891
7892
7893
7894
MEMORY
Add
3456
3457
Store
3458
Add
7889
7890
Store
7891
Compare
3458
7891
Print
CONTROL UNIT
ARITHMETIC LOGIC UNIT
INPUT
OUTPUT

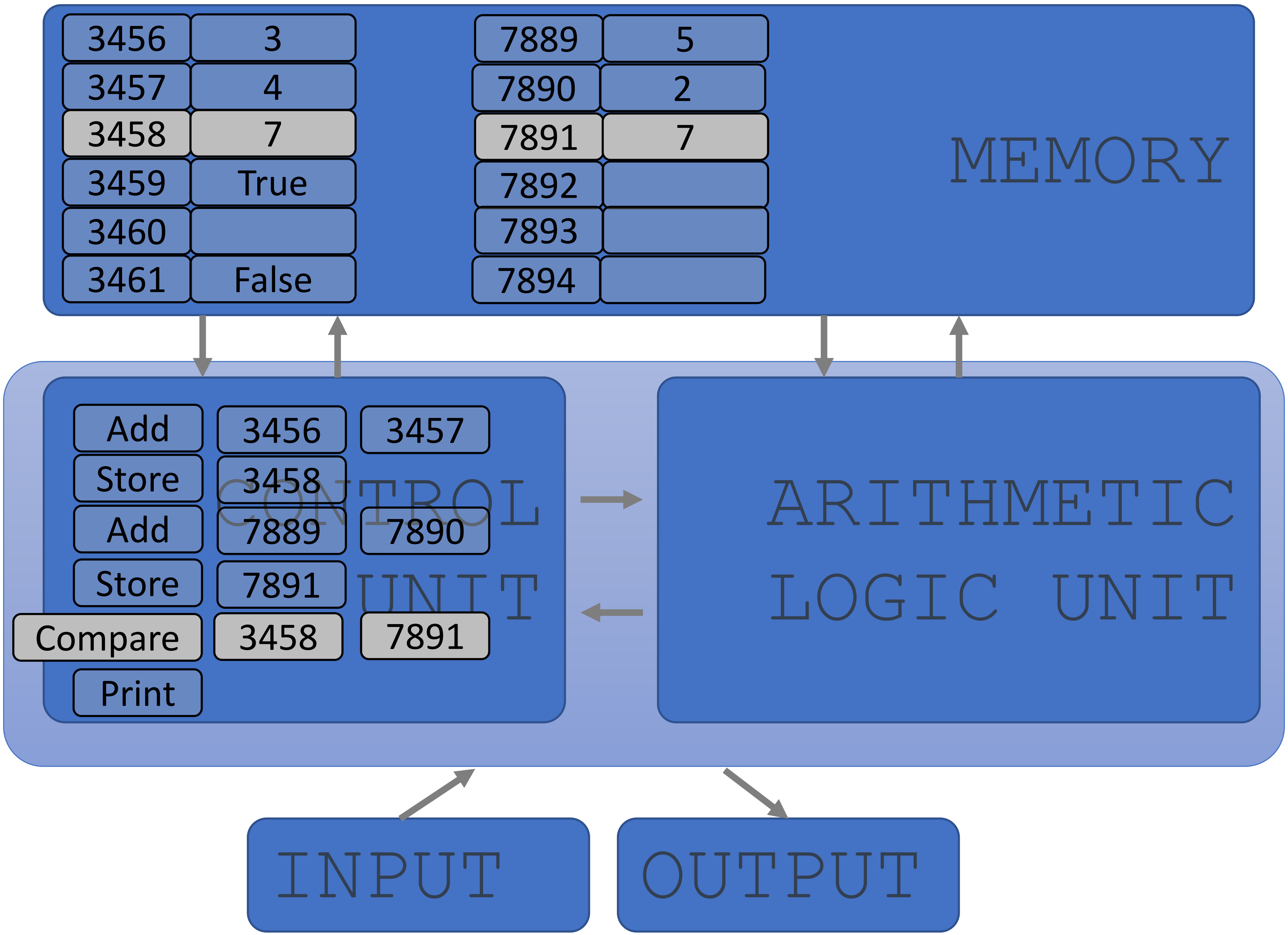


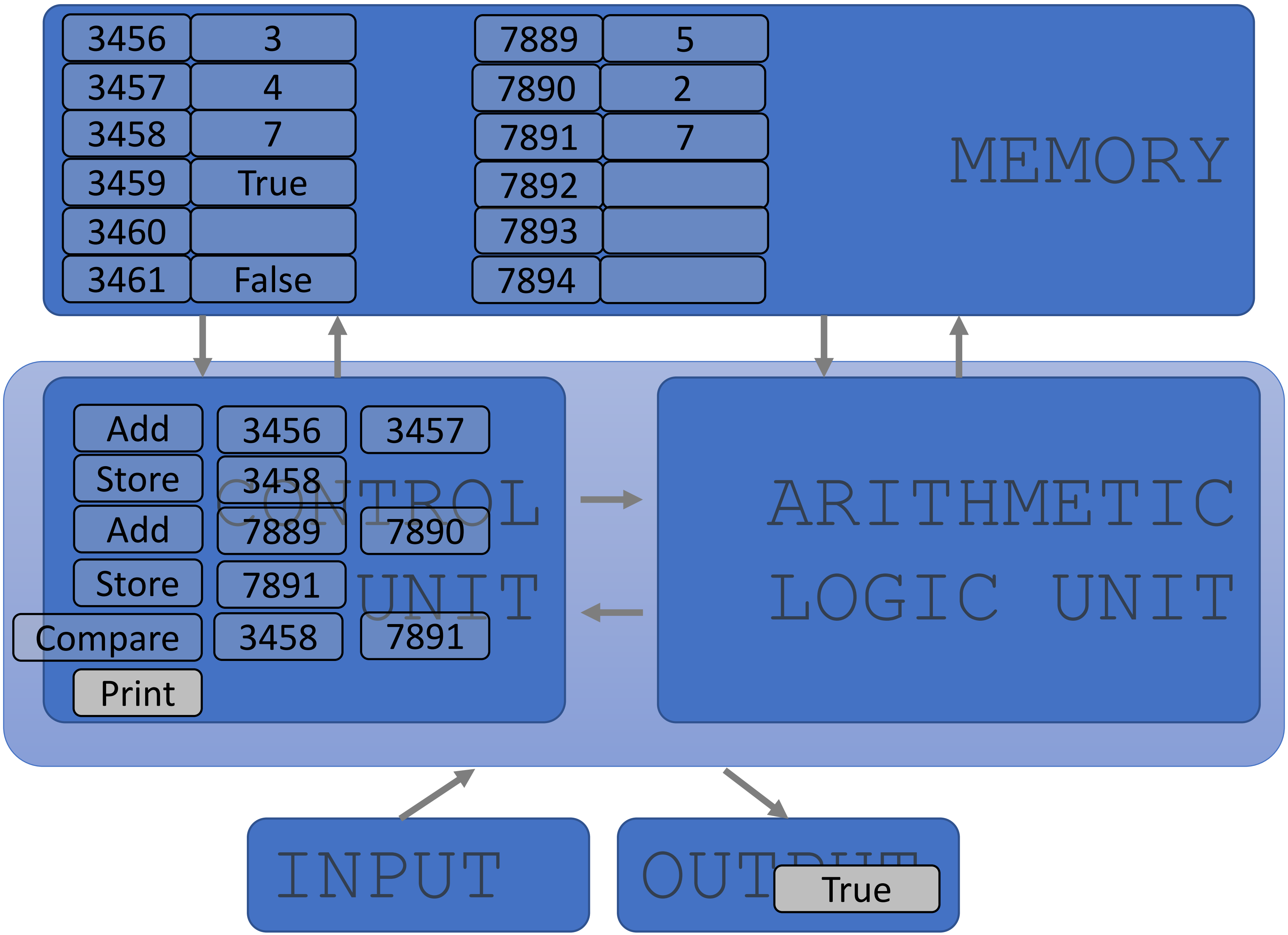
3456
3
3457
4
3458
3459
True
3460
3461
False
7889
5
7890
2
7891
7892
7893
7894
MEMORY
CONTROL UNIT
Add
3456
3457
Store
3458
Add
7889
7890
Store
7891
Compare
3458
7891
Print
ARITHMETIC LOGIC UNIT
INPUT
OUTPUT


3456
3
3457
4
3458
7
3459
True
3460
3461
False
7889
5
7890
2
7891
7892
7893
7894
MEMORY
Add
3456
3457
Store
3458
Add
7889
7890
Store
7891
Compare
3458
7891
Print
UNIT
ARITHMETIC
LOGIC UNIT
INPUT
OUTPUT


3456
3
3457
4
3458
7
3459
True
3460
3461
False
7889
5
7890
2
7891
7892
7893
7894
MEMORY
Add
3456
3457
Store
3458
CONTROL
Add
7889
7890
Store
7891
UNIT
Compare
3458
7891
Print
ARITHMETIC
LOGIC UNIT
INPUT
OUTPUT

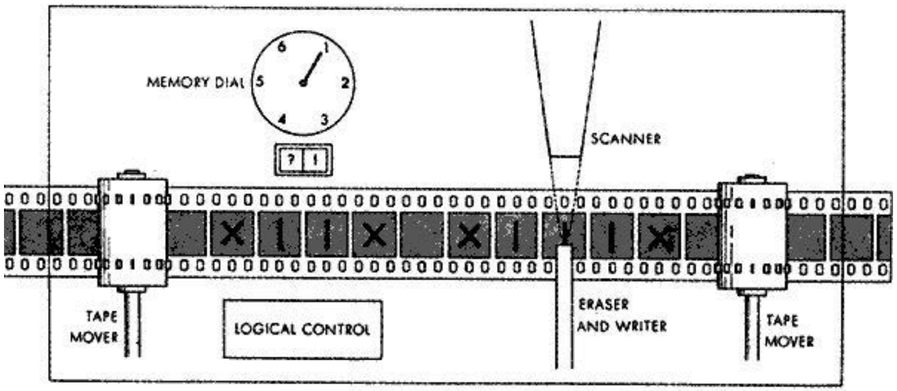

3456
3
3457
4
3458
7
3459
True
3460
3461
False
7889
5
7890
2
7891
7
7892
7893
7894
MEMORY
Add
3456
3457
Store
3458
Add
7889
7890
Store
7891
Compare
3458
7891
Print
CONTROL UNIT
ARITHMETIC LOGIC UNIT
INPUT
OUTPUT


3456
3
3457
4
3458
7
3459
True
3460
3461
False
7889
5
7890
2
7891
7
7892
7893
7894
MEMORY
Add
3456
3457
Store
3458
Add
7889
7890
Store
7891
Compare
3458
7891
Print
CONTROL UNIT
ARITHMETIC LOGIC UNIT
INPUT
OUTPUT


MEMORY
3456
3
3457
4
3458
7
3459
True
3460
3461
False
7889
5
7890
2
7891
7
7892
7893
7894
CONTROL
UNIT
Add
3456
3457
Store
3458
Add
7889
7890
Store
7891
Compare
3458
7891
Print
ARITHMETIC
LOGIC UNIT
INPUT
OUTPUT
True

BASIC PRIMITIVES

- Turing showed that you can **compute anything** with a very simple machine with only 6 primitives: left, right, print, scan, erase, no op



- Real programming languages have
  - More convenient set of primitives
  - Ways to combine primitives to **create new primitives**
- Anything computable in one language is computable in any other programming language

ASPECTS of LANGUAGES

- **Primitive constructs**
  - English: words
  - Programming language: numbers, strings, simple operators

ASPECTS of LANGUAGES

- **Syntax**
  - English: `"cat dog boy"` → not syntactically valid
    `"cat hugs boy"` → syntactically valid
  - Programming language: `"hi"5` → not syntactically valid
    `"hi"*5` → syntactically valid

ASPECTS of LANGUAGES

- **Static semantics**: which syntactically valid strings have meaning
  - English: `"I are hungry"` → syntactically valid
    but static semantic error
  - PL: `"hi"+5` → syntactically valid
    but static semantic error

ASPECTS of LANGUAGES

- **Semantics**: the meaning associated with a syntactically correct string of symbols with no static semantic errors
- English: can have many meanings "`The chicken is ready to eat.`"
- Programs have only one meaning
- **But the meaning may not be what programmer intended**

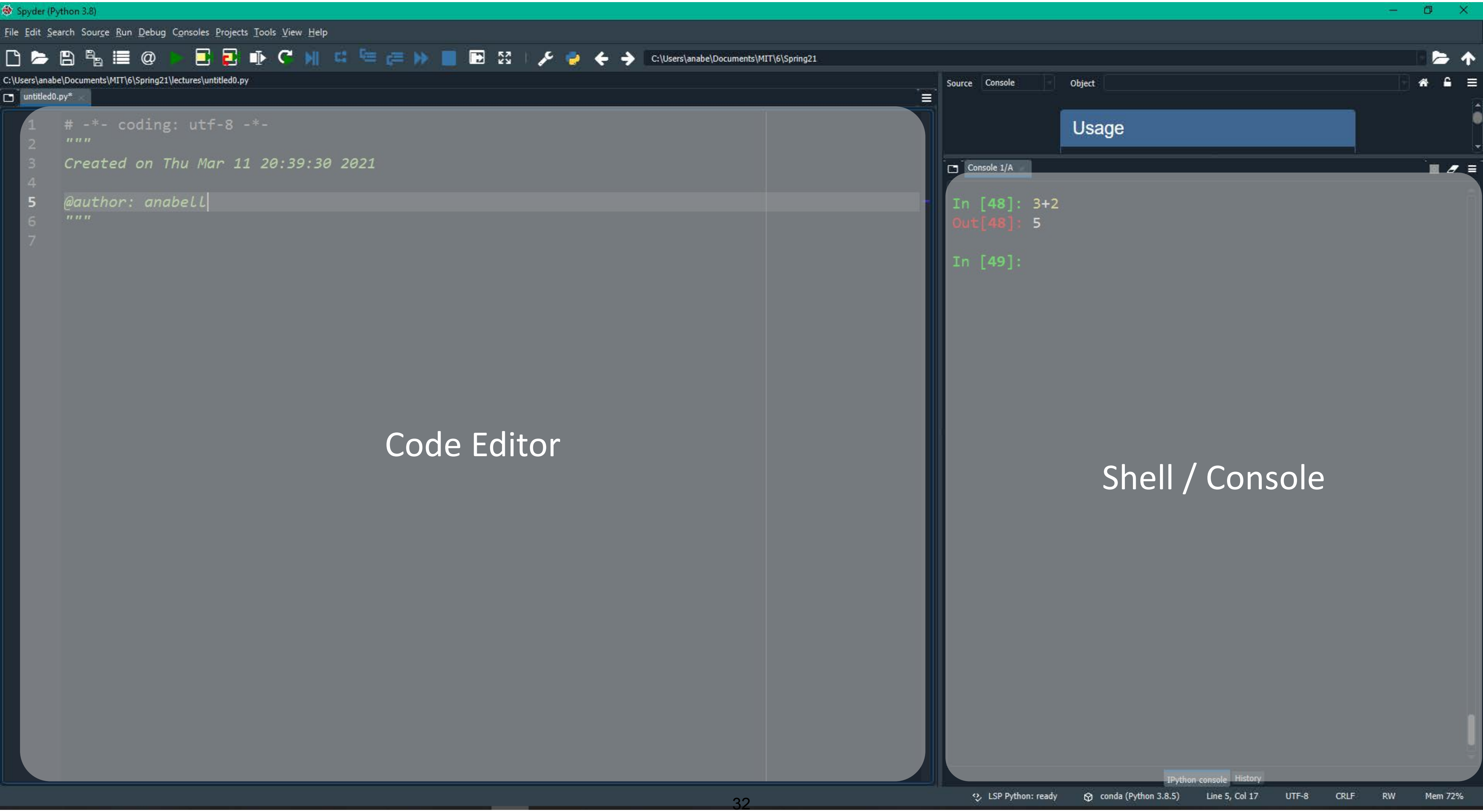
WHERE THINGS GO WRONG

- **Syntactic errors**
  - Common and easily caught
- **Static semantic errors**
  - Some languages check for these before running program
  - Can cause unpredictable behavior
- No linguistic errors, but **different meaning than what programmer intended**
  - Program crashes, stops running
  - Program runs forever
  - Program gives an answer, but it’s wrong!

PYTHON PROGRAMS

- A **program** is a sequence of definitions and commands
  - Definitions **evaluated**
  - Commands **executed** by Python interpreter in a shell
- **Commands** (statements) instruct interpreter to do something
- Can be typed directly in a **shell** or stored in a **file** that is read into the shell and evaluated
  - Problem Set 0 will introduce you to these in Anaconda

PROGRAMMING ENVIRONMENT: ANACONDA



OBJECTS

- Programs manipulate **data objects**
- Objects have a **type** that defines the kinds of things programs can do to them
  - `30`
    - Is a number
    - We can add/sub/mult/div/exp/etc
  - `'Ana'`
    - Is a sequence of characters (aka a string)
    - We can grab substrings, but we can’t divide it by a number

OBJECTS

- **Scalar** (cannot be subdivided)
  - Numbers: 8.3, 2
  - Truth value: True, False

- **Non-scalar** (have internal structure that can be accessed)
  - Lists
  - Dictionaries
  - Sequence of characters: "abc"

SCALAR OBJECTS

- `int` – represent **integers**, ex. `5, -100`
- `float` – represent **real numbers**, ex. `3.27, 2.0`
- `bool` – represent **Boolean** values `True` and `False`
- `NoneType` – **special** and has one value, `None`
- Can use `type()` to see the type of an object

```
>>> type(5)
int
>>> type(3.0)
float
```

what you write into the Python shell

what shows after hitting enter

int

0, 1, 2, …
300, 301 …
-1, -2, -3, …
-400, -401, …

float

0.0, …, 0.21, …
1.0, …, 3.14, …
-1.22, …, -500.0 , …

bool

True
False

NoneType

None

YOU TRY IT!

- In your console, find the type of:
  - `1234`
  - `8.99`
  - `9.0`
  - `True`
  - `False`

TYPE CONVERSIONS (CASTING)

- Can **convert object of one type to another**
  - `float(3)` casts the int `3` to float `3.0`
  - `int(3.9)` casts (note the truncation!) the float `3.9` to int `3`
- Some operations perform implicit casts
  - `round(3.9)` returns the int `4`

YOU TRY IT!

- In your console, find the type of:
  - `float(123)`
  - `round(7.9)`
  - `float(round(7.2))`
  - `int(7.2)`
  - `int(7.9)`

EXPRESSIONS

- **Combine objects and operators** to form expressions
  - 3+2
  - 5/3
- An expression has a **value**, which has a type
  - 3+2 has value 5 and type int
  - 5/3 has value 1.666667 and type float
- Python evaluates expressions and stores the value. It doesn't store expressions!

- Syntax for a simple expression
  `<object> <operator> <object>`

BIG IDEA

Replace complex expressions by ONE value

Work systematically to evaluate the expression.

EXAMPLES

- `>>> 3+2`
- `5`
- `>>>` $\boxed{\texttt{(4+2)*6-1}}$
- `35`
- `>>>` $\boxed{\texttt{type((4+2)*6-1)}}$
- `int`
- `>>>` $\boxed{\texttt{float((4+2)*6-1)}}$
- `35.0`

Do computations left to right – like in math!

Do computations inside parens first, left to right

Take care about what operations you are doing

YOU TRY IT!

- In your console, find the values of the following expressions:
  - `(13-4) / (12*12)`
  - `type(4*3)`
  - `type(4.0*3)`
  - `int(1/2)`

OPERATORS on `int` and `float`

- `i+j` → the **sum**
- `i-j` → the **difference**
- `i*j` → the **product**

if both are ints, result is int
if either or both are floats, result is float

- `i/j` → **division**

result is always a float

- `i//j` → **floor division**

What is type of output?

- `i%j` → the **remainder** when `i` is divided by `j`
- `i**j` → `i` to the **power** of `j`

SIMPLE OPERATIONS

- Parentheses tell Python to do these operations first
  - Like math!
- **Operator precedence** without parentheses

`**`

`*  /  %` executed left to right, as appear in expression

`+  -` executed left to right, as appear in expression

SO MANY OBJECTS, what to do with them?!

a = 2

temp = 100.4

go = True

b = -0.3

flag = False

x = 123

n = 17

small = 0.001

VARIABLES

- Computer science variables are **different** than math variables
- **Math variables**
  - Abstract
  - Can **represent many values**

```
a + 2 = b - 1
```

```
x * x = y
```

x represents all square roots

- **CS variables**
  - Is bound to **one single value** at a given time
  - Can be bound to an expression
    (but expressions evaluate to one value!)

```
a = b + 1
```

```
m = 10
F = m*9.98
```

one variable

one value

BINDING VARIABLES to VALUES

- In CS, the equal sign is an **assignment**
  - One value to one variable name
  - Equal sign is **not equality**, not “solve for x”
- An assignment binds a value to a name

variable value

```
pi = 355/113
```

- **Step 1:** Compute the value on the **right hand side** (the VALUE)
  - Value stored in computer memory
- **Step 2:** Store it (bind it) to the **left hand side** (the VARIABLE)
  - Retrieve value associated with name by invoking the name (typing it out)

YOU TRY IT!

- Which of these are allowed in Python? Type them in the console to check.
  - `x = 6`
  - `6 = x`
  - `x*y = 3+4`
  - `xy = 3+4`

ABSTRACTING EXPRESSIONS

- Why **give names** to values of expressions?
  - To **reuse names** instead of values
  - Makes code easier to read and modify
- Choose variable names wisely
  - Code needs to read
  - Today, tomorrow, next year
  - By you and others
  - You'll be fine if you stick to letters, underscores, don't start with a number

```
#Compute approximate value for pi
pi = 355/113
radius = 2.2
area = pi*(radius**2)
circumference = pi*(radius*2)
```

comments start with a # and are not part of code executed – used to tell others what your code is doing

an assignment
* expression on right
* variable name on left

WHAT IS BEST CODE STYLE?

```
#do calculations
a = 355/113 *(2.2**2)
c = 355/113 *(2.2*2)
```

meh

```
p = 355/113
r = 2.2
#multiply p with r squared
a = p*(r**2)
#multiply p with r times 2
c = p*(r*2)
```

ok

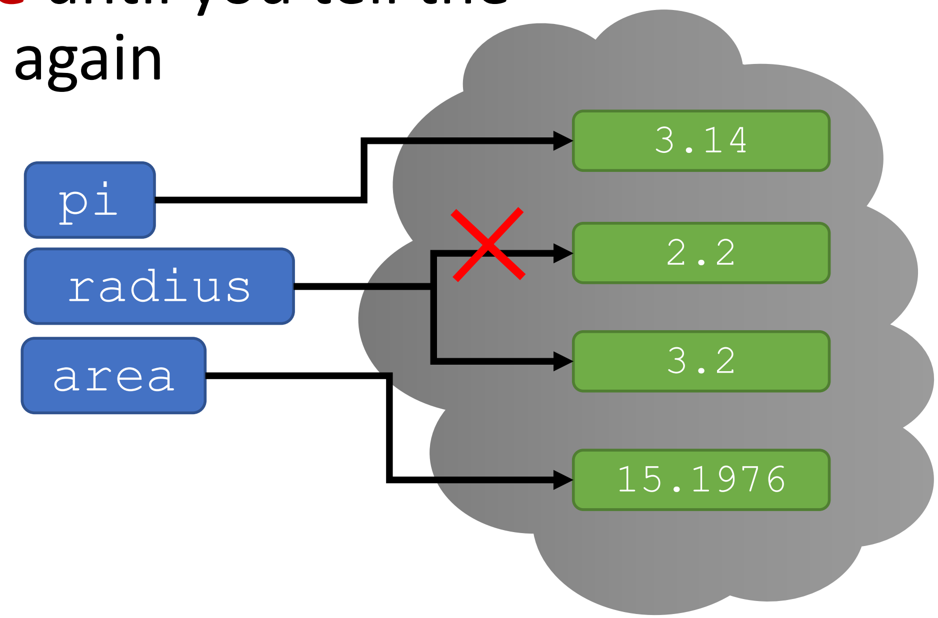
```
#calculate area and circumference of a circle
#using an approximation for pi
pi = 355/113
radius = 2.2
area = pi*(radius**2)
circumference = pi*(radius*2)
```

best

CHANGE BINDINGS

- Can **re-bind** variable names using new assignment statements
- Previous value may still stored in memory but lost the handle for it
- Value for **area does not change** until you tell the computer to do the calculation again

```
pi = 3.14
radius = 2.2
area = pi*(radius**2)
radius = radius+1
```



BIG IDEA

Lines are evaluated one after the other

No skipping around, yet.
We'll see how lines can be skipped/repeated later.

YOU TRY IT!

- These 3 lines are executed in order. What are the values of `meters` and `feet` variables at each line in the code?

```
meters = 100
feet = 3.2808 * meters
meters = 200
```

ANSWER:

Let's use PythonTutor to figure out what is going on

- Follow along with this Python Tutor LINK

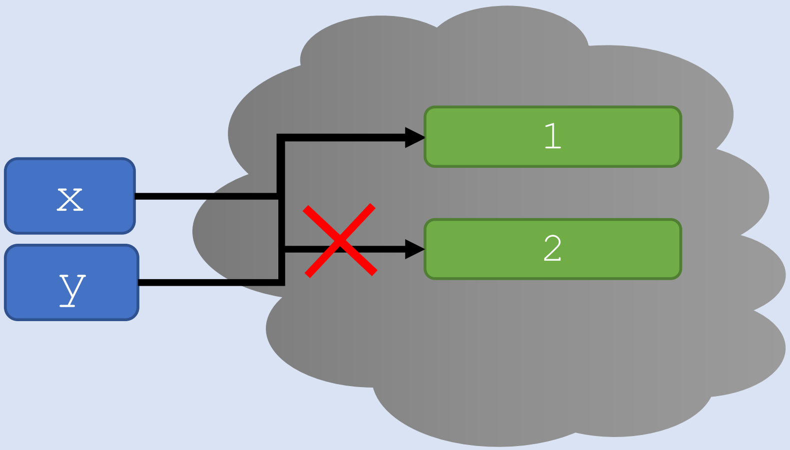
Where did we tell Python to (re)calculate feet?

YOU TRY IT!

- Swap values of x and y without binding the numbers directly. Debug (aka fix) this code.

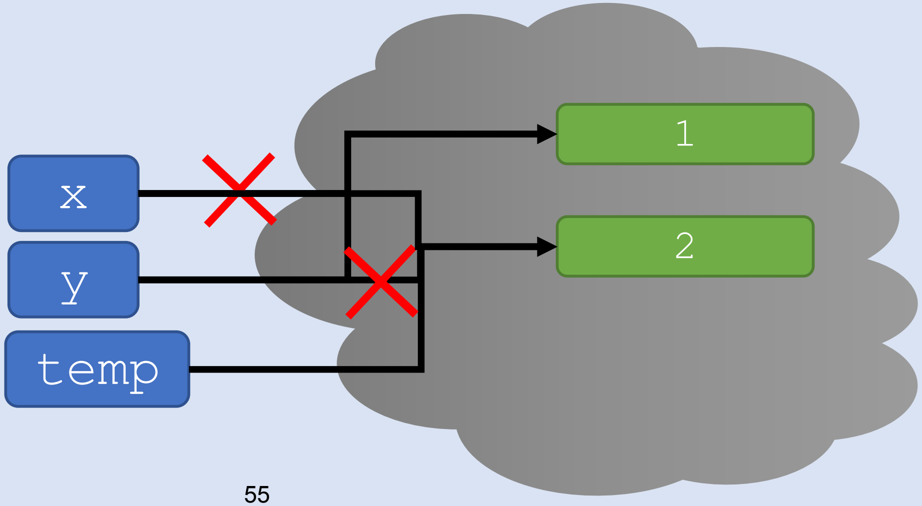
```
x = 1
y = 2

y = x
x = y
```



- Python Tutor to the rescue?

ANSWER:



SUMMARY

- Objects
  - Objects in memory have **types**.
  - Types tell Python what **operations** you can do with the objects.
  - **Expressions evaluate to one value** and involve objects and operations.
  - Variables bind names to objects.
  - = sign is an assignment, for ex. `var = type(5*4)`
- Programs
  - Programs only **do what you tell them to do**.
  - Lines of code are executed **in order**.
  - Good variable names and comments help you **read code later**.

6.100L Introduction to Computer Science and Programming Using Python
Fall 2022

STRINGS, INPUT/OUTPUT, and BRANCHING

(download slides and .py files to follow along)

6.100L Lecture 2

Ana Bell

RECAP

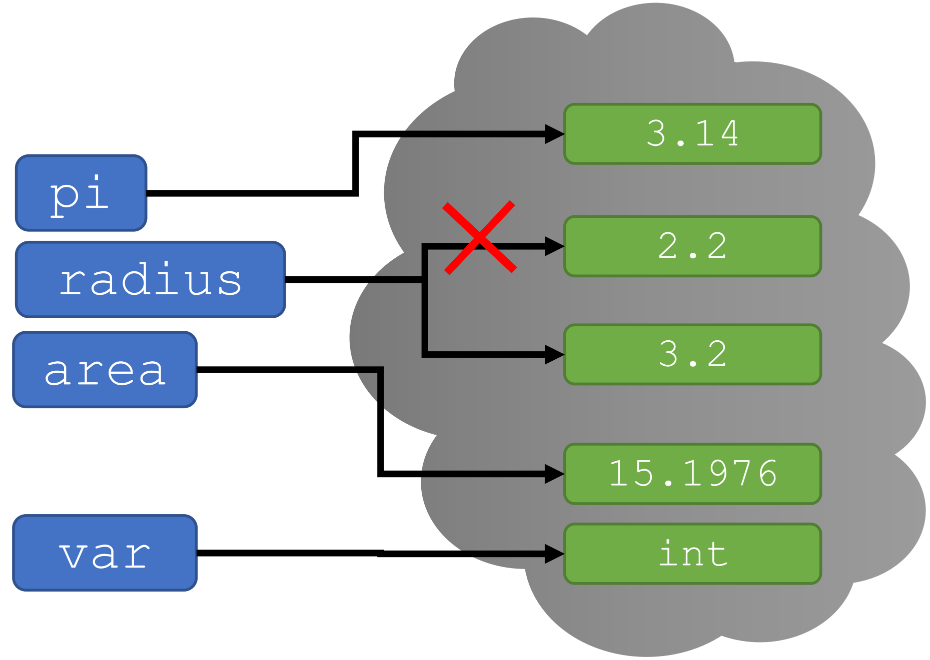
```
pi = 3.14

radius = 2.2

area = pi*(radius**2)

radius = radius+1


var = type(5*4)
```



- Objects
  - Objects in memory have **types**.
  - Types tell Python what **operations** you can do with the objects.
  - **Expressions evaluate to one value** and involve objects and operations.
  - Variables bind names to objects.
  - = sign is an assignment, for ex. `var = type(5*4)`

- Programs
  - Programs only **do what you tell them to do**.
  - Lines of code are executed **in order**.
  - Good variable names and comments help you **read code later**.

STRINGS

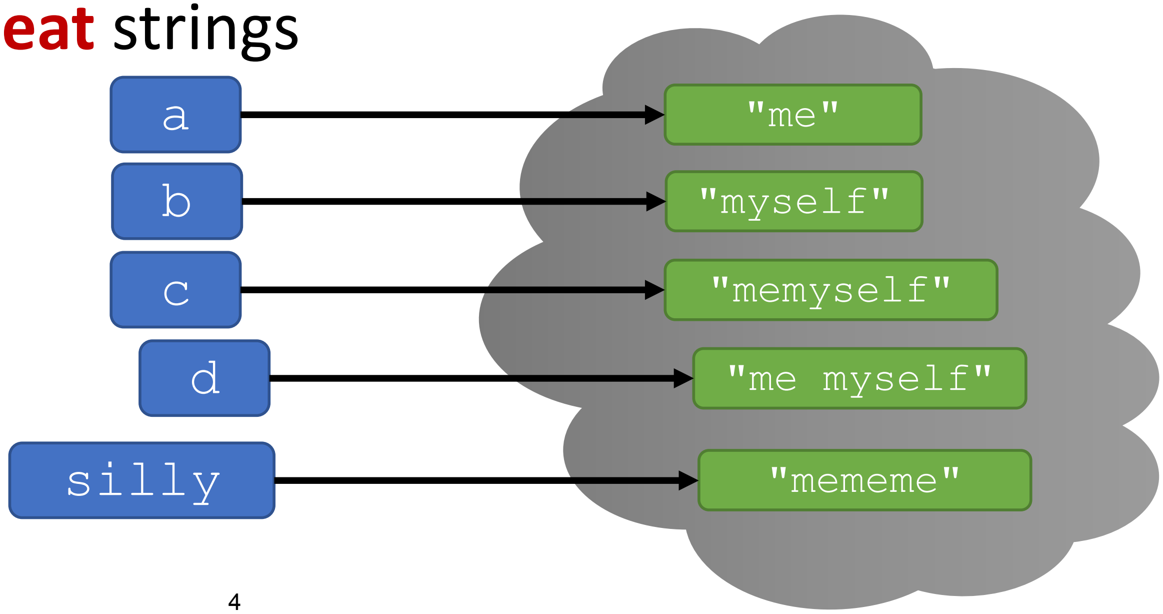
STRINGS

- Think of a `str` as a **sequence** of case sensitive characters
  - Letters, special characters, spaces, digits
- Enclose in **quotation marks or single quotes**
  - Just be consistent about the quotes

```
a = "me"
z = 'you'
```

- **Concatenate** and **repeat** strings

```
b = "myself"
c = a + b
d = a + " " + b
silly = a * 3
```



4

YOU TRY IT!

What's the value of s1 and s2?

- ```
  b = ":"
  c = ")"
  s1 = b + 2*c
  ```
- ```
  f = "a"
  g = " b"
  h = "3"
  s2 = (f+g)*int(h)
  ```

STRING OPERATIONS

- `len()` is a function used to retrieve the **length** of a string in the parentheses

```
s = "abc"
```

`len(s)` → evaluates to 3

`chars =` $\boxed{\texttt{len(s)}}$ Expression that evaluates to 3

SLICING to get ONE CHARACTER IN A STRING

- Square brackets used to perform **indexing** into a string to get the value at a certain index/position

```
s = "abc"
```

index: 0 1 2 ← indexing always starts at 0

index: -3 -2 -1 ← index of last element is len(s) - 1 or -1

`s[0]` → evaluates to "a"

`s[1]` → evaluates to "b"

`s[2]` → evaluates to "c"

`s[3]` → trying to index out of bounds, error

`s[-1]` → evaluates to "c"

`s[-2]` → evaluates to "b"

`s[-3]` → evaluates to "a"

SLICING to get a SUBSTRING

- Can **slice** strings using `[start:stop:step]`
- Get characters at **start**
  up to and including **stop-1**
  taking every **step** characters

This is confusing as you are starting out :(
Can't go wrong with explicitly giving start, stop, end every time.

- If give two numbers, [`start:stop`], `step=1` by default
- If give one number, you are back to indexing for the character at one location (prev slide)
- You can also omit numbers and leave just colons (try this out!)

8

SLICING EXAMPLES

- Can **slice** strings using `[start:stop:step]`
- Look at step first. +ve means go left-to-right
  -ve means go right-to-left

`s = "abcdefgh"`

index:	0	1	2	3	4	5	6	7
index:	-8	-7	-6	-5	-4	-3	-2	-1

If unsure what some command does, try it out in your console!

`s[3:6]` → evaluates to `"def"`, same as `s[3:6:1]`

`s[3:6:2]` → evaluates to `"df"`

`s[:]` → evaluates to `"abcdefgh"`, same as `s[0:len(s):1]`

`s[::-1]` → evaluates to `"hgfedcba"`

`s[4:1:-2]` → evaluates to `"ec"`

YOU TRY IT!

```
s = "ABC d3f ghi"

s[3:len(s)-1]
s[4:0:-1]
s[6:3]
```

10

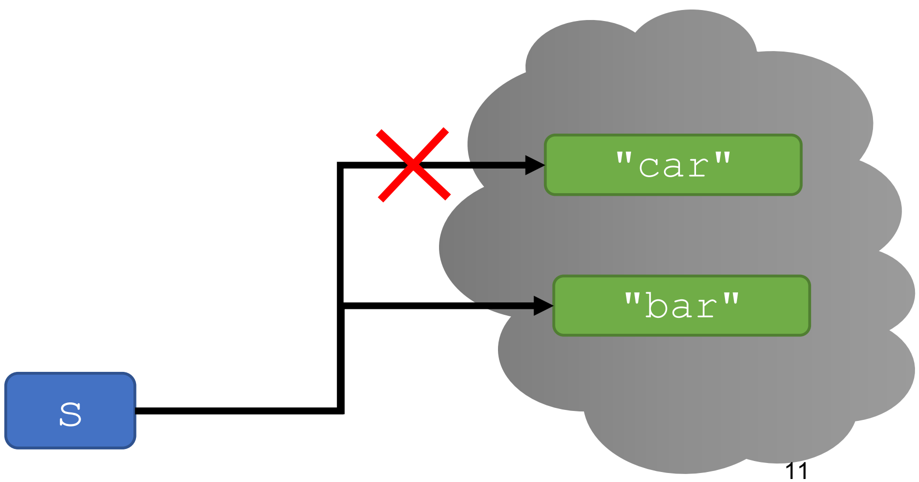
IMMUTABLE STRINGS

- Strings are "**immutable**" – cannot be modified
- You can create **new objects** that are versions of the original one
- Variable name can only be bound to one object

```
s = "car"
```

```
s[0] = 'b'
```
→ gives an error

```
s = 'b'+s[1:len(s)]
```
→ is allowed,
s bound to new object



11

BIG IDEA

If you are wondering "what happens if"…

Just try it out in the console!

INPUT/OUTPUT

PRINTING

- Used to **output** stuff to console

```
In [11]: 3+2
Out[11]: 5
```

“Out” tells you it’s an interaction within the shell only

- Command is `print`

```
In [12]: print(3+2)
5
```

No “Out” means it is actually shown to a user, apparent when you edit/run files

- Printing many objects in the same command
  - Separate objects using commas to output them separated by spaces
  - Concatenate strings together using + to print as single object
  - ```
    a = "the"
    b = 3
    c = "musketeers"
    print(a, b, c)
    print(a + str(b) + c)
    ```

Every piece being concatenated must be a string

14
    ```

INPUT

- `x = input(s)`
  - Prints the value of the string `s`
  - User types in something and hits enter
  - That value is assigned to the variable `x`
- **Binds that value to a variable**

```
text = input("Type anything: ")
print(5*text)
```

SHELL:

```
Type anything:
```

And it waits for characters and Enter to be hit

INPUT

- `x = input(s)`
  - Prints the value of the string `s`
  - User types in something and hits enter
  - That value is assigned to the variable `x`
- **Binds that value to a variable**

“howdy”

```
text = input("Type anything: ")
print(5*text)
```

SHELL:

```
Type anything: howdy
```

16

INPUT

- `x = input(s)`
  - Prints the value of the string `s`
  - User types in something and hits enter
  - That value is assigned to the variable `x`
- **Binds that value to a variable**

```
text = input("Type anything: ")
print(5*text)
```

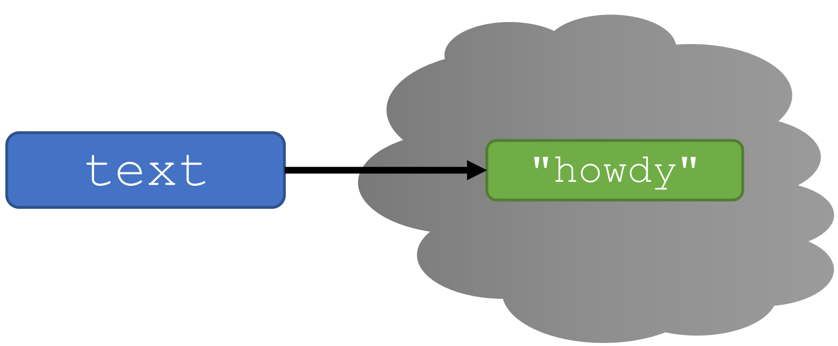
SHELL:

```
Type anything: howdy
```

INPUT

- `x = input(s)`
  - Prints the value of the string `s`
  - User types in something and hits enter
  - That value is assigned to the variable `x`
- **Binds that value to a variable**

```
text = input("Type anything: ")
print(5*text)
```



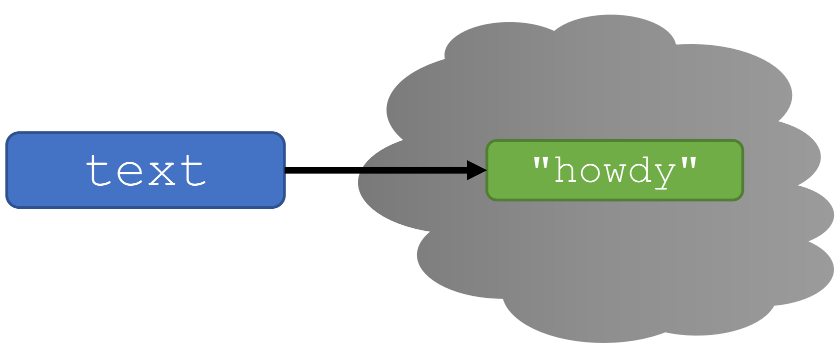
SHELL:

```
Type anything: howdy
```

INPUT

- `x = input(s)`
  - Prints the value of the string `s`
  - User types in something and hits enter
  - That value is assigned to the variable `x`
- **Binds that value to a variable**

```
text = input("Type anything: ")
print(5*text)
```



SHELL:

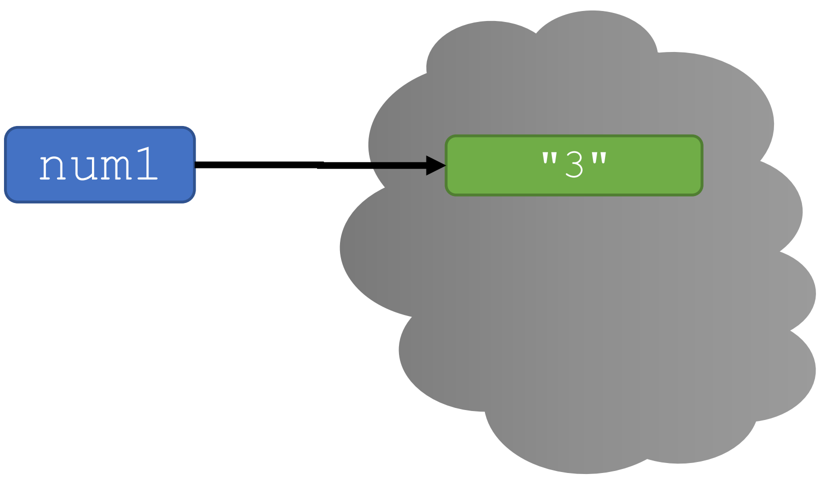
```
Type anything: howdy
howdyhowdyhowdyhowdyhowdy
```

INPUT

- `input` always returns an **str,** must cast if working with numbers

```
num1 = input("Type a number: ")
```
"3"

```
print(5*num1)
num2 = int(input("Type a number: "))
print(5*num2)
```



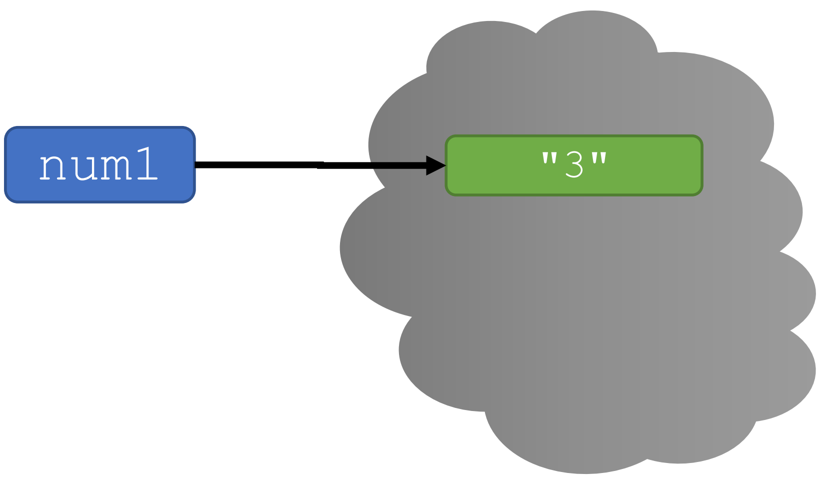
SHELL:

```
Type a number: 3
```

INPUT

- `input` always returns an **str,** must cast if working with numbers

```
num1 = input("Type a number: ")
print(5*num1)
num2 = int(input("Type a number: "))
print(5*num2)
```



SHELL:

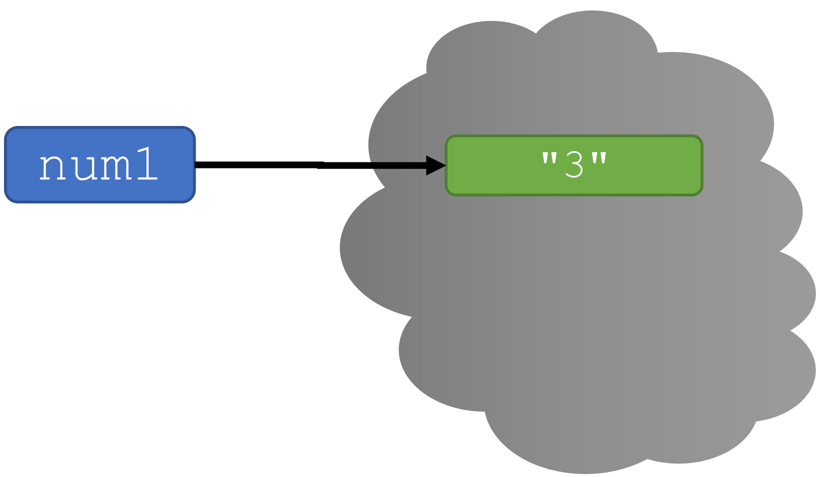
```
Type a number: 3
33333
```

INPUT

- `input` always returns an **str,** must cast if working with numbers

```
num1 = input("Type a number: ")
print(5*num1)
num2 = int(input("Type a number: "))
print(5*num2)
```

"3"



SHELL:

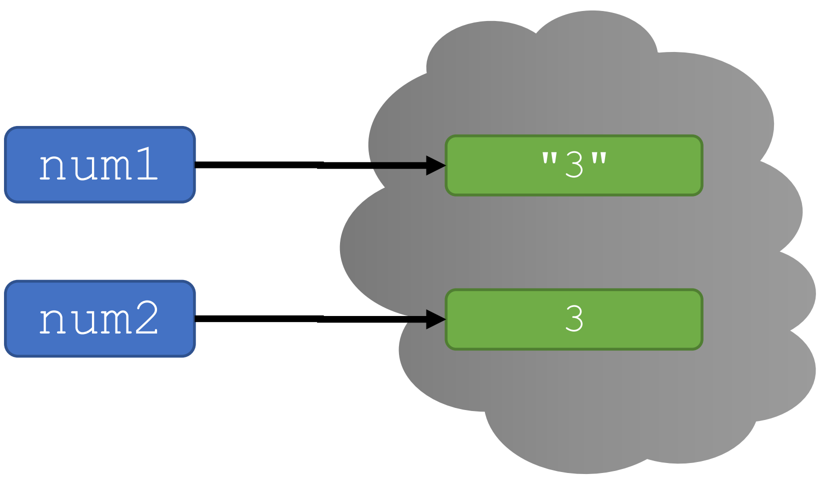
```
Type a number: 3
33333
Type a number: 3
```

INPUT

- `input` always returns an **str,** must cast if working with numbers

```
num1 = input("Type a number: ")
print(5*num1)
num2 = int(input("Type a number: "))
print(5*num2)
```

3



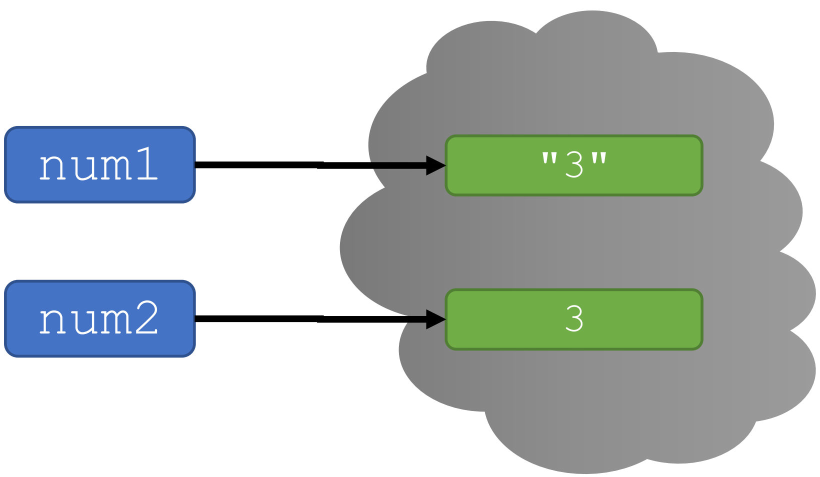
SHELL:

```
Type a number: 3
33333
Type a number: 3
```

INPUT

- `input` always returns an **str,** must cast if working with numbers

```
num1 = input("Type a number: ")
print(5*num1)
num2 = int(input("Type a number: "))
print(5*num2)
```



SHELL:

```
Type a number: 3
33333
Type a number: 3
15
```

YOU TRY IT!

- Write a program that
  - Asks the user for a verb
  - Prints “I can _ better than you” where you replace _ with the verb.
  - Then prints the verb 5 times in a row separated by spaces.
  - For example, if the user enters `run`, you print:

```
I can run better than you!
run run run run run
```

AN IMPORTANT ALGORITHM: NEWTON'S METHOD

- Finds roots of a polynomial
  - E.g., find g such that $f(g, x) = g^3 - x = 0$
- Algorithm uses successive approximation
  - next_guess = guess - $\frac{f(guess)}{f'(guess)}$
- Partial code of algorithm that gets input and finds next guess

```
#Try Newton Raphson for cube root
x = int(input('What x to find the cube root of? '))
g = int(input('What guess to start with? '))
print('Current estimate cubed = ', g**3)
```

f(g) f'(g)

```
next_g = g - ((g**3 - x)/(3*g**2))
print('Next guess to try = ', next_g)
```

F-STRINGS

- Available starting with Python 3.6
- Character `f` followed by a **formatted string literal**
  - Anything that can be appear in a normal string literal
  - Expressions bracketed by curly braces { }
- Expressions in curly braces evaluated at runtime, automatically converted to strings, and concatenated to the string preceding them

Introduces an extra space

```
num = 3000
fraction = 1/3
print(num*fraction, 'is', fraction*100, '% of', num)
print(num*fraction, 'is', str(fraction*100) + '% of', num)
print(f'{num*fraction} is {fraction*100}% of {num}')
```

expressions

27

BIG IDEA

Expressions can be placed anywhere.

Python evaluates them!

CONDITIONS for BRANCHING

BINDING VARIABLES and VALUES

- In CS, there are two **notions of equal**
  - Assignment and Equality test
- `variable = value`
  - **Change the stored value** of variable to value
  - Nothing for us to solve, computer just does the action
- `some_expression == other_expression`
  - A **test for equality**
  - No binding is happening
  - Expressions are replaced by values and computer just does the comparison
  - Replaces the **entire line** with `True` or `False`

COMPARISON OPERATORS

- `i` and `j` are variable names
  - They can be of type ints, float, strings, etc.
- Comparisons below evaluate to the type **Boolean**
  - The Boolean type only has 2 values: `True` and `False`

`i > j`

`i >= j`

`i < j`

`i <= j`

`i == j` → **equality** test, `True` if `i` is the same as `j`

`i != j` → **inequality** test, `True` if `i` not the same as `j`

With strings, be careful about case sensitivity: 'March' != 'march'

31

LOGICAL OPERATORS on bool

- `a` and `b` are variable names (with Boolean values)

`not a` → `True` if `a` is `False`
`False` if `a` is `True`

`a and b` → `True` if both are `True`

`a or b` → `True` if either or both are `True`

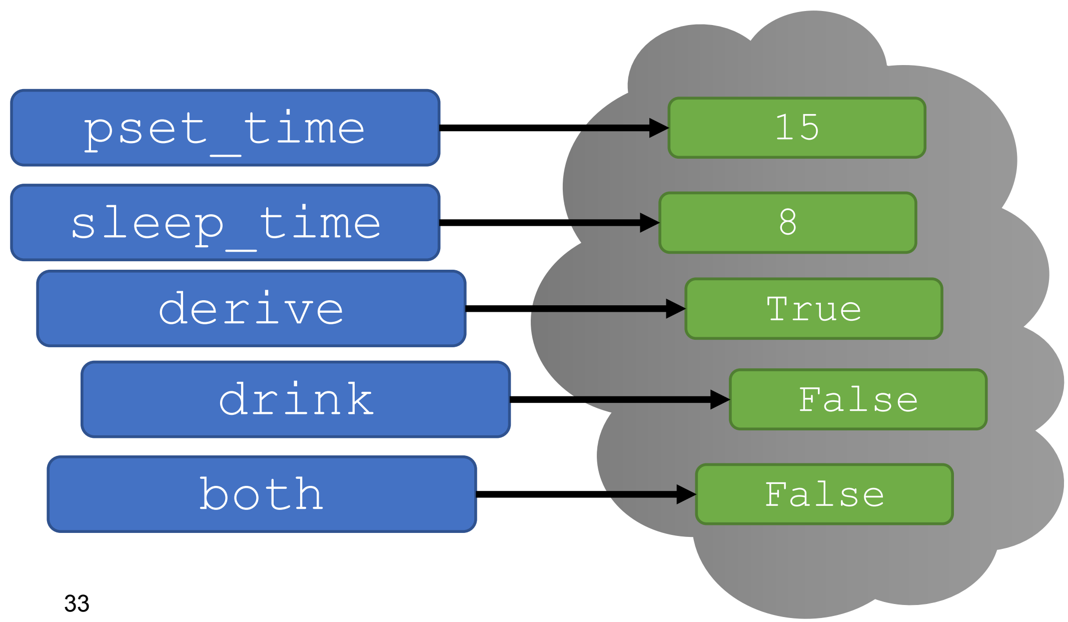
A	B	A and B	A or B
True	True	True	True
True	False	False	True
False	True	False	True
False	False	False	False

COMPARISON EXAMPLE

```
pset_time = 15
sleep_time = 8
print(sleep_time > pset_time)
derive = True
drink = False
both = drink and derive
print(both)
```

Prints the boolean False

Prints the boolean False



YOU TRY IT!

- Write a program that
  - Saves a secret number in a variable.
  - Asks the user for a number guess.
  - Prints a bool `False` or `True` depending on whether the guess matches the secret.

WHY bool?

- When we get to flow of control, i.e. branching to different expressions based on values, we need a way of knowing if a condition is true
- E.g., if $\boxed{\text{something is true}}$, $\boxed{\text{do this}}$, otherwise $\boxed{\text{do that}}$

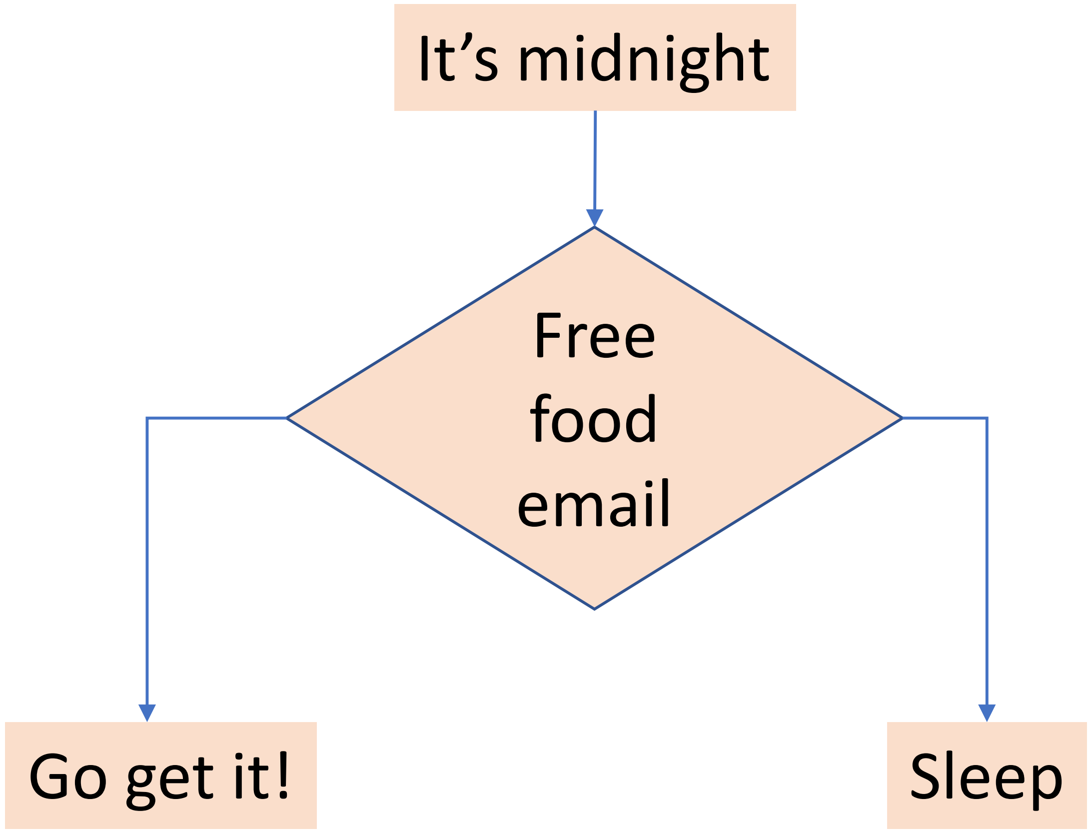
Boolean

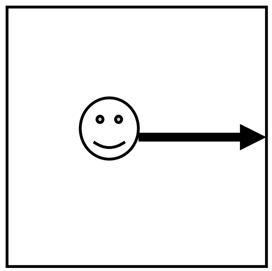
Some commands

Some other commands

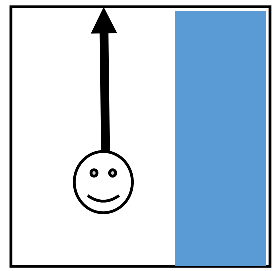
35

INTERESTING ALGORITHMS INVOLVE DECISIONS

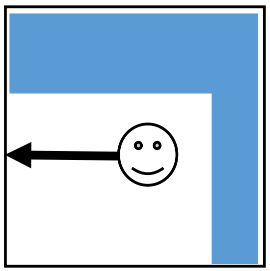




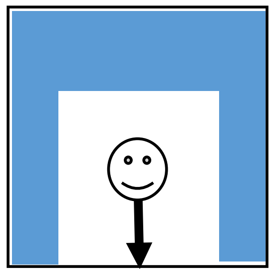
If right clear,
go right



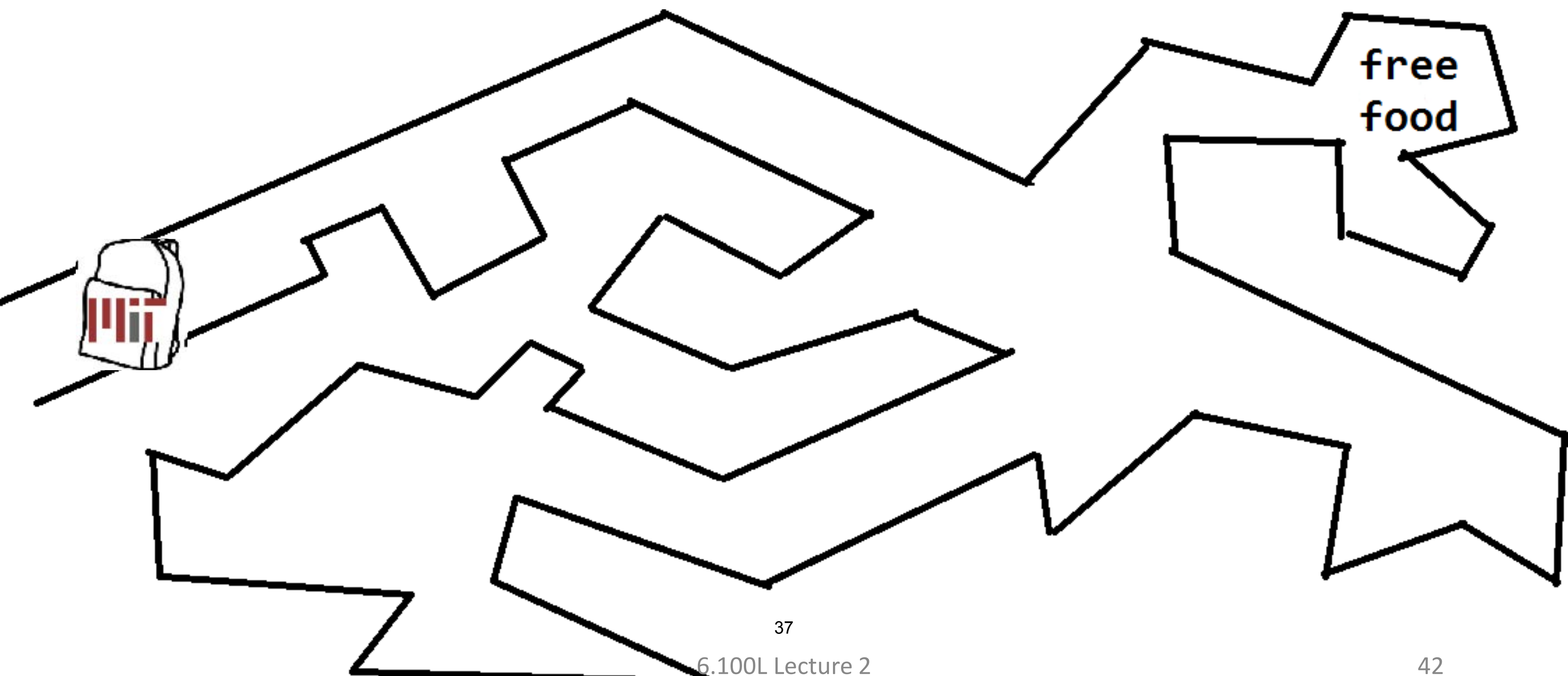
If right blocked,
go forward



If right and
front blocked,
go left



If right , front,
left blocked,
go back



BRANCHING IN PYTHON

```
if <condition>:
    <code>
    <code>
    ...
<rest of program>
```

- <condition> has a value True or False
- **Indentation matters** in Python!
- Do code within if block if condition is True

38

BRANCHING IN PYTHON

```
if <condition>:
    <code>
    <code>
    ...
<rest of program>
```

```
if <condition>:
    <code>
    <code>
    ...
else:
    <code>
    <code>
    ...
<rest of program>
```

- `<condition>` has a value `True` or `False`
- **Indentation matters** in Python!
- Do code within if block when condition is `True` **or** code within else block when condition is `False` 39

BRANCHING IN PYTHON

```
if <condition>:
    <code>
    <code>
    ...
<rest of program>
```

```
if <condition>:
    <code>
    <code>
    ...
else:
    <code>
    <code>
    ...
<rest of program>
```

```
if <condition>:
    <code>
    <code>
    ...
elif <condition>:
    <code>
    <code>
    ...
elif <condition>:
    <code>
    <code>
    ...
<rest of program>
```

- `<condition>` has a value `True` or `False`
- **Indentation matters** in Python!
- Run the **first block** whose corresponding `<condition>` is `True`

BRANCHING IN PYTHON

```
if <condition>:
    <code>
    <code>
    ...
<rest of program>
```

```
if <condition>:
    <code>
    <code>
    ...
else:
    <code>
    <code>
    ...
<rest of program>
```

```
if <condition>:
    <code>
    <code>
    ...
elif <condition>:
    <code>
    <code>
    ...
elif <condition>:
    <code>
    <code>
    ...
<rest of program>
```

```
if <condition>:
    <code>
    <code>
    ...
elif <condition>:
    <code>
    <code>
    ...
else:
    <code>
    <code>
    ...
<rest of program>
```

- `<condition>` has a value `True` or `False`
- **Indentation matters** in Python!
- Run the **first block** whose corresponding `<condition>` is `True`. The else block runs when no conditions were `True`

BRANCHING EXAMPLE

```
pset_time = ???
sleep_time = ???
if (pset_time + sleep_time) > 24:
    print("impossible!")
elif (pset_time + sleep_time) >= 24:
    print("full schedule!")
else:
    leftover = abs(24-pset_time-sleep_time)
    print(leftover,"h of free time!")
print("end of day")
```

Condition that evaluates to a Boolean

This indented code executed if line above is True

This indented code executed if line above is True and the if condition is False

This else block runs only if previous conditions were all False

YOU TRY IT!

- Semantic structure matches visual structure
- Fix this buggy code (hint, it has bad indentation)!

```
x = int(input("Enter a number for x: "))
y = int(input("Enter a different number for y: "))
if x == y:
    print(x,"is the same as",y)
print("These are equal!")
```

INDENTATION and NESTED BRANCHING

- Matters in Python
- How you **denote blocks of code**

```
x = float(input("Enter a number for x: "))
y = float(input("Enter a number for y: "))
if x == y:
    print("x and y are equal")
    if y != 0:
        print("therefore, x / y is", x/y)
elif x < y:
    print("x is smaller")
else:
    print("y is smaller")
print("thanks!")
```

5	5	0
5	0	0
True	False	True
<-		<-
True		False
<-		
	False	
	<-	
<-	<-	<-

BIG IDEA

Practice will help you build a mental model of how to trace the code

Indentation does a lot of the work for you!

YOU TRY IT!

- What does this code print with
  - y = 2
  - y = 20
  - y = 11
- What if `if x <= y:` becomes `elif x <= y:` ?

```
answer = ''
x = 11
if x == y:
    answer = answer + 'M'
if x >= y:
    answer = answer + 'i'
else:
    answer = answer + 'T'
print(answer)
```

YOU TRY IT!

- Write a program that
  - Saves a secret number.
  - Asks the user for a number guess.
  - Prints whether the guess is too low, too high, or the same as the secret.

BIG IDEA

Debug early, debug often.

Write a little and test a little.

Don't write a complete program at once. It introduces too many errors.

Use the Python Tutor to step through code when you see something unexpected!

SUMMARY

- Strings provide a new data type
  - They are **sequences of characters**, the **first one at index 0**
  - They can be indexed and sliced
- Input
  - Done with the **`input`** command
  - Anything the user inputs is **read as a string object**!
- Output
  - Is done with the **`print`** command
  - Only objects that are printed in a .py code file will be **visible in the shell**
- Branching
  - Programs execute **code blocks** when conditions are true
  - In an `if-elif-elif`… structure, the **first condition that is True** will be executed
  - **Indentation matters** in Python!

ITERATION

(download slides and .py files to follow along)

6.100L Lecture 3

Ana Bell

LAST LECTURE RECAP

- Strings provide a new data type
  - They are **sequences of characters**, the **first one at index 0**
  - They can be indexed and sliced
- Input
  - Done with the **`input`** command
  - Anything the user inputs is **read as a string object**!
- Output
  - Is done with the **`print`** command
  - Only objects that are printed in a .py code file will be **visible in the shell**
- Branching
  - Programs execute **code blocks** when conditions are true
  - In an `if-elif-elif`… structure, the **first condition that is True** will be executed
  - **Indentation matters** in Python!

BRANCHING RECAP

```
if <condition>:
    < code >
    < code >
    ...
```

```
if <condition>:
    < code >
    < code >
    ...
else:
    < code >
    < code >
    ...
```

```
if <condition>:
    < code >
    < code >
    ...
elif <condition>:
    < code >
    < code >
    ...
elif <condition>:
    < code >
    < code >
    ...
```

```
if <condition>:
    < code >
    < code >
    ...
elif <condition>:
    < code >
    < code >
    ...
else:
    < code >
    < code >
    ...
```

- `<condition>` has a value `True` or `False`
- Evaluate the **first block** whose corresponding `<condition>` is `True`
    - A block is started by an `if` statement
- **Indentation matters** in Python!

- If you keep going right, you are stuck in the same spot forever
- To exit, take a chance and go the opposite way

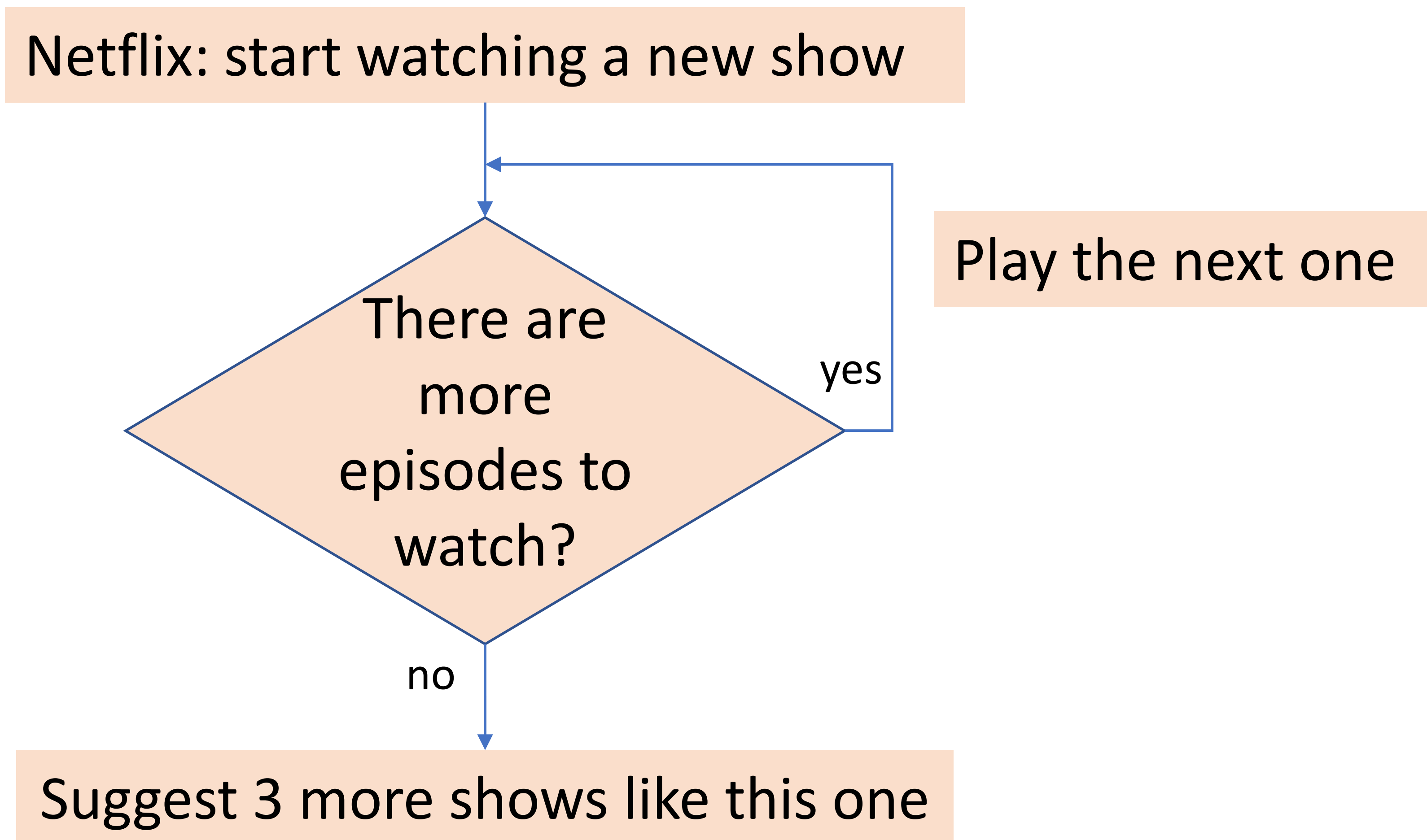

```
if <exit right>:
    <set background to woods_background>
    if <exit right>:
        <set background to woods_background>
        if <exit right>:
            <set background to woods_background>
            and so on and on and on...
        else:
            <set background to exit_background>
    else:
        <set background to exit_background>
else:
    <set background to exit_background>
```

- If you keep going right, you are stuck in the same spot forever
- To exit, take a chance and go the opposite way


```
while <exit_right>:
    <set background to woods_background>
    <ask user which way to go>
<set background to exit_background>
```

`while` LOOPS

BINGE ALL EPISODES OF ONE SHOW



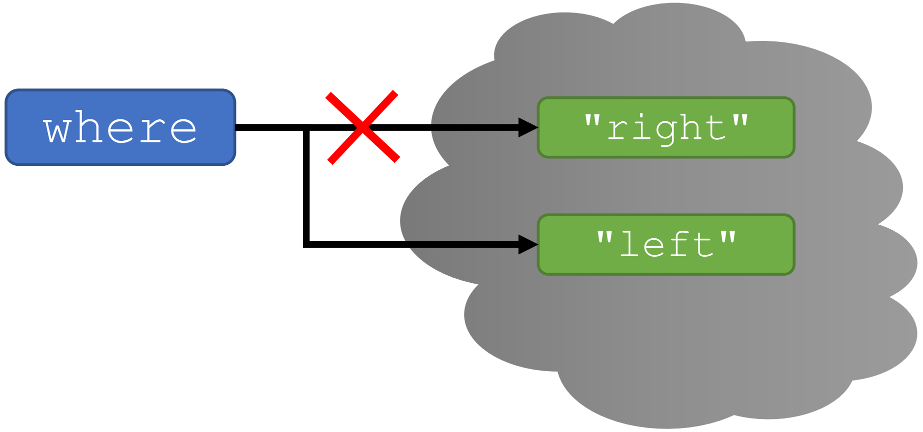
CONTROL FLOW: while LOOPS

```
while <condition>:
      <code>
      <code>
      ...
```

- <condition> **evaluates to a Boolean**
- If <condition> is `True`, **execute all the steps inside** the while code block
- **Check** <condition> again
- **Repeat** until <condition> is `False`
- If <condition> is never `False`, then will loop forever!!

while LOOP EXAMPLE

```
You are in the Lost Forest.
************
************
   ☺
************
************
Go left or right?
```



PROGRAM:

```python
where = input("You're in the Lost Forest. Go left or right? ")
while where == "right":
    where = input("You're in the Lost Forest. Go left or right? ")
print("You got out of the Lost Forest!")
```

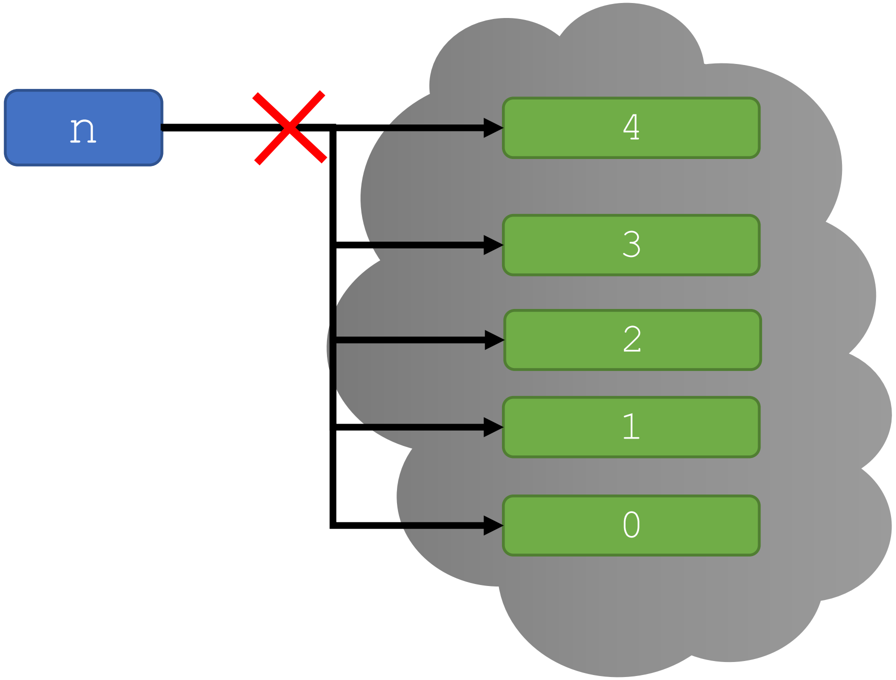
YOU TRY IT!

- What is printed when you type "RIGHT"?

```
where = input("Go left or right? ")
while where == "right":
    where = input("Go left or right? ")
print("You got out!")
```

while LOOP EXAMPLE

```
n = int(input("Enter a non-negative integer: "))
while n > 0:
    print('x')
    n = n-1
```



while LOOP EXAMPLE

```
n = int(input("Enter a non-negative integer: "))
while n > 0:
    print('x')
    n = n-1
```

What happens without this last line?
Try it!

- To terminate:
  - Hit CTRL-c or CMD-c in the shell
  - Click the red square in the shell

YOU TRY IT!

- Run this code and stop the infinite loop in your IDE

```
while True:
    print("noooooo")
```

BIG IDEA

`while` loops can repeat code inside indefinitely!

Sometimes they need your intervention to end the program.

YOU TRY IT!

- Expand this code to show a sad face when the user entered the while loop more than 2 times.
- Hint: use a variable as a counter

```
where = input("Go left or right? ")
while where == "right":
    where = input("Go left or right? ")
print("You got out!")
```

CONTROL FLOW: while LOOPS

- Iterate through **numbers in a sequence**

Set loop variable outside while loop

```
n = 0
while n < 5:
    print(n)
    n = n+1
```

Test loop variable in condition

Increment loop variable inside while loop

n = n+1
equivalent to
n += 1

A COMMON PATTERN

- Find 4!
- `i` is our loop variable
- `factorial` keeps track of the product

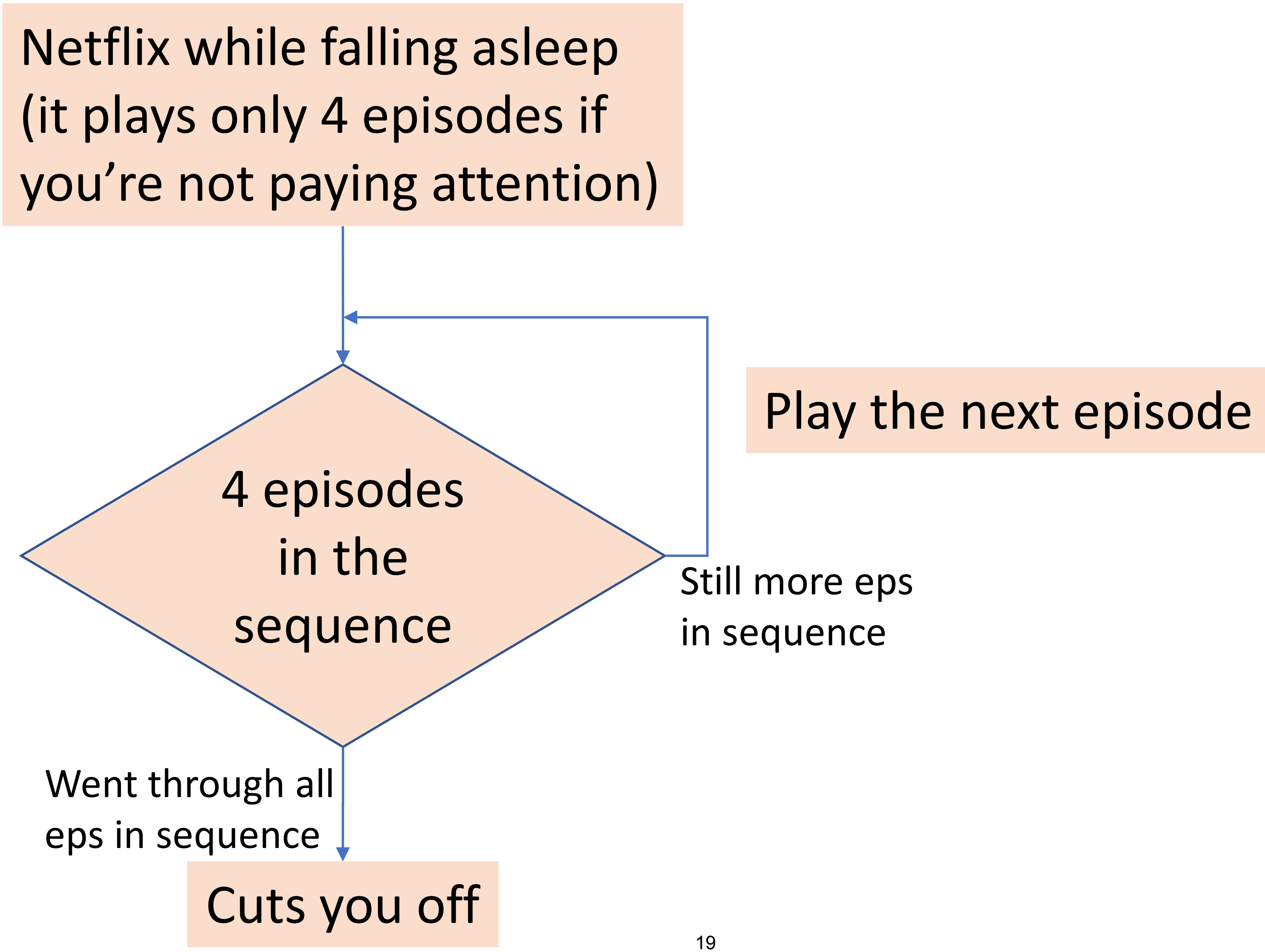
```
x = 4
i = 1
factorial = 1
while i <= x:
    factorial *= i
    i += 1
print(f'{x} factorial is {factorial}')
```

Set loop variable outside while loop
Initialize the factorial product to 1
Test loop variable in condition
Keep a running product (eq to factorial = factorial*i)
Increment loop variable inside while loop (eq to i = i+1)

- Python Tutor LINK

`for` LOOPS

ARE YOU STILL WATCHING?



CONTROL FLOW: `while` and `for` LOOPS

- Iterate through **numbers in a sequence**

```
# very verbose with while loop
n = 0
while n < 5:
    print(n)
    n = n+1


# shortcut with for loop
for n in range(5):
    print(n)
```

STRUCTURE of `for` LOOPS

```
for <variable> in <sequence of values>:
    <code>
    ...
```

- **Each time through the loop**, `<variable>` takes a value
- First time, `<variable>` is the **first value in sequence**
- Next time, `<variable>` gets the **second value**
- etc. until `<variable>` runs out of values

A COMMON SEQUENCE of VALUES

```
for <variable> in range(<some_num>):
    <code>
    <code>
    ...
```

Sequence is 0 then 1 then 2 then 3 then 4

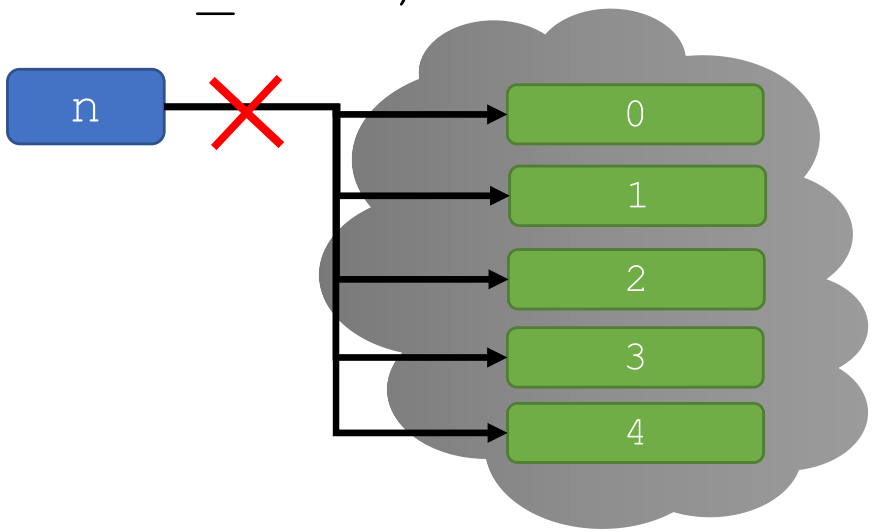
```
for n in range(5):
    print(n)
```

- **Each time through the loop**, <variable> takes a value
- First time, <variable> **starts at 0**
- Next time, <variable> gets the value **1**
- Then, <variable> gets the value **2**
- ...
- etc. until <variable> gets **some_num -1**

A COMMON SEQUENCE of VALUES

```
for <variable> in range(<some_num>):
    <code>
    <code>
    ...
```

```
for n in range(5):
    print(n)
```



- **Each time through the loop**, <variable> takes a value
- First time, <variable> **starts at 0**
- Next time, <variable> gets the value **1**
- Then, <variable> gets the value **2**
- ...
- etc. until <variable> gets **some_num -1**

range

- Generates a **sequence** of ints, following a pattern
- `range(start, stop, step)`
  - `start`: first int generated
  - `stop`: controls last int generated (go up to but not including this int)
  - `step`: used to generate next int in sequence
- A lot like what we saw for **slicing**
- Often omit start and step
  - e.g., `for i in range(4):`
    - `start` defaults to 0
    - `step` defaults to 1
  - e.g., `for i in range(3,5):`
    - `step` defaults to 1

Remember strings? It had a similar syntax, but with colons not commas and square brackets not parentheses.

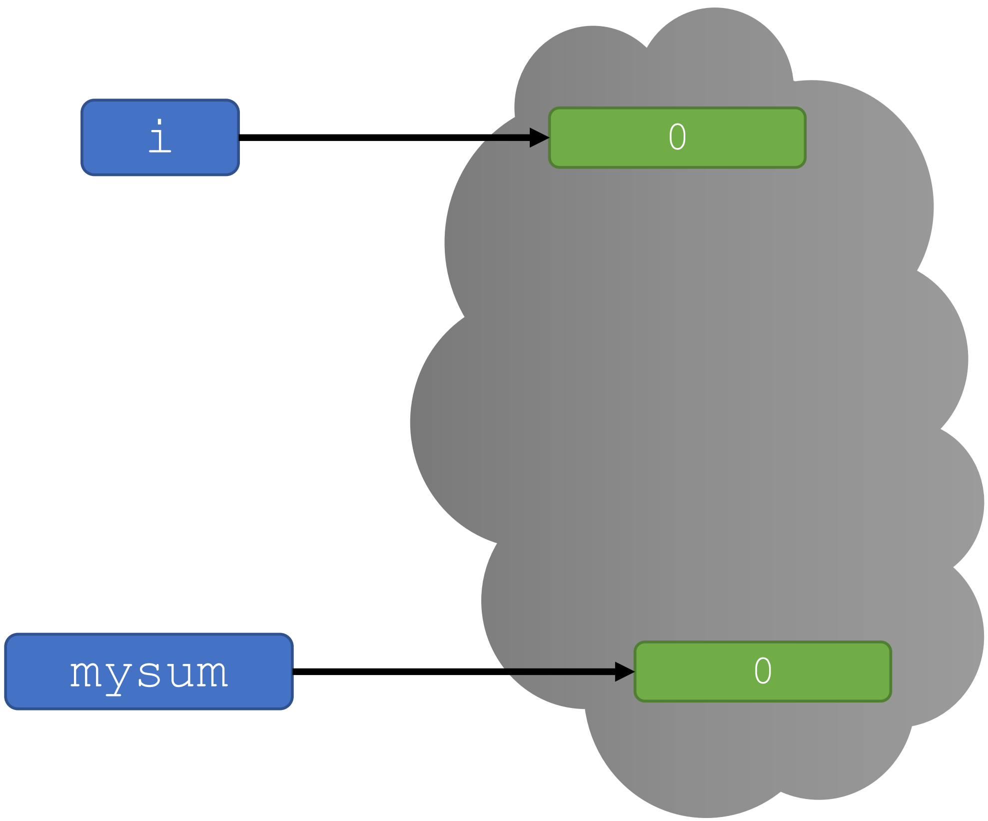
YOU TRY IT!

- What do these print?
- ```
  for i in range(1,4,1):
      print(i)
  ```
- ```
  for j in range(1,4,2):
      print(j*2)
  ```
- ```
  for me in range(4,0,-1):
      print("$"*me)
  ```
  ```

RUNNING SUM

- `mysum` is a variable to store the **running sum**
- `range(10)` makes `i` be 0 then 1 then 2 then … then 9

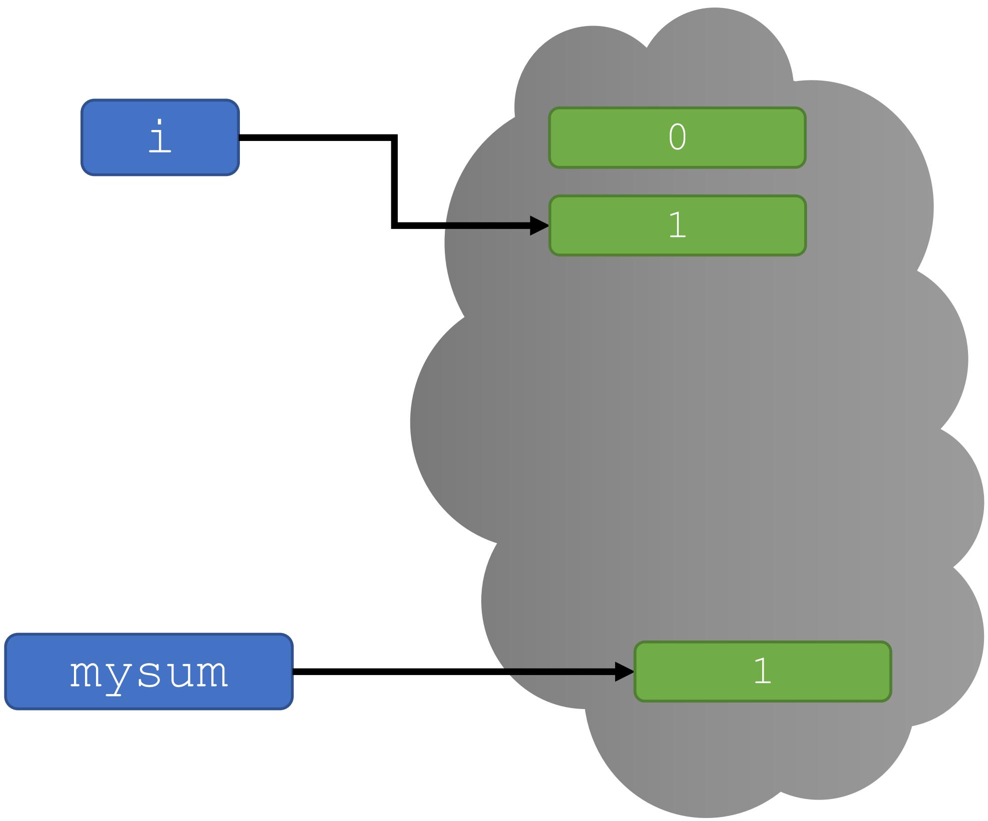
```
mysum = 0
for i in range(10):
    mysum += i
print(mysum)
```



RUNNING SUM

- `mysum` is a variable to store the **running sum**
- `range(10)` makes `i` be 0 then 1 then 2 then … then 9

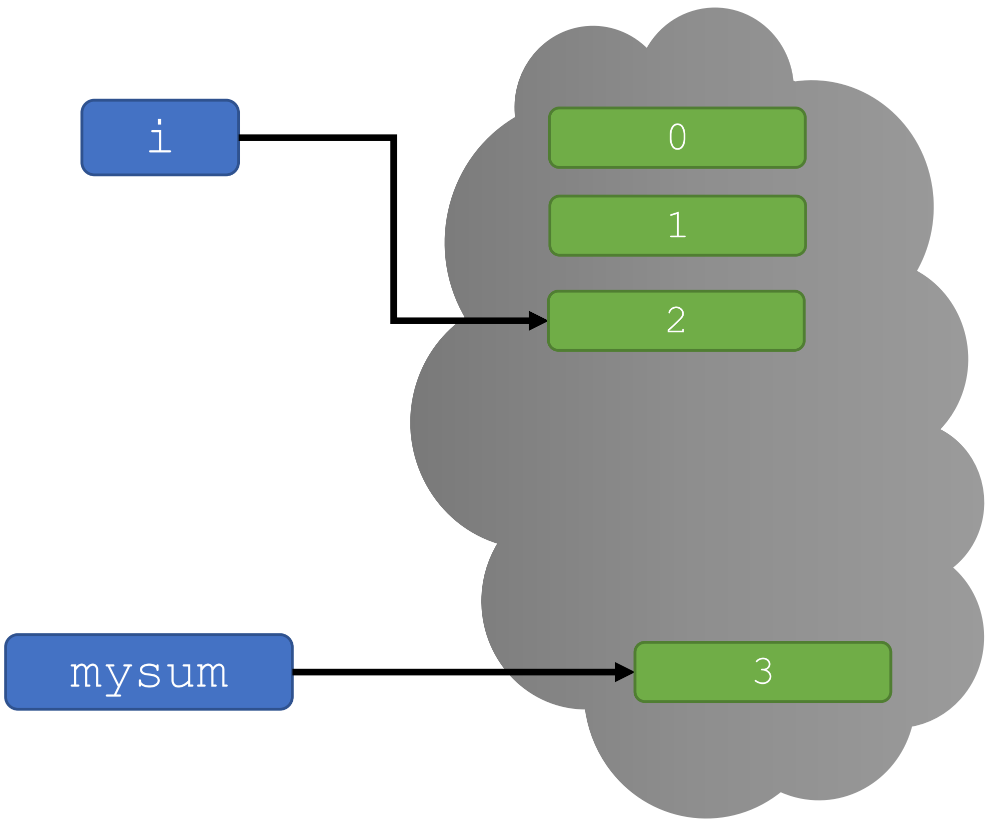
```
mysum = 0
for i in range(10):
    mysum += i
print(mysum)
```



RUNNING SUM

- `mysum` is a variable to store the **running sum**
- `range(10)` makes `i` be 0 then 1 then 2 then … then 9

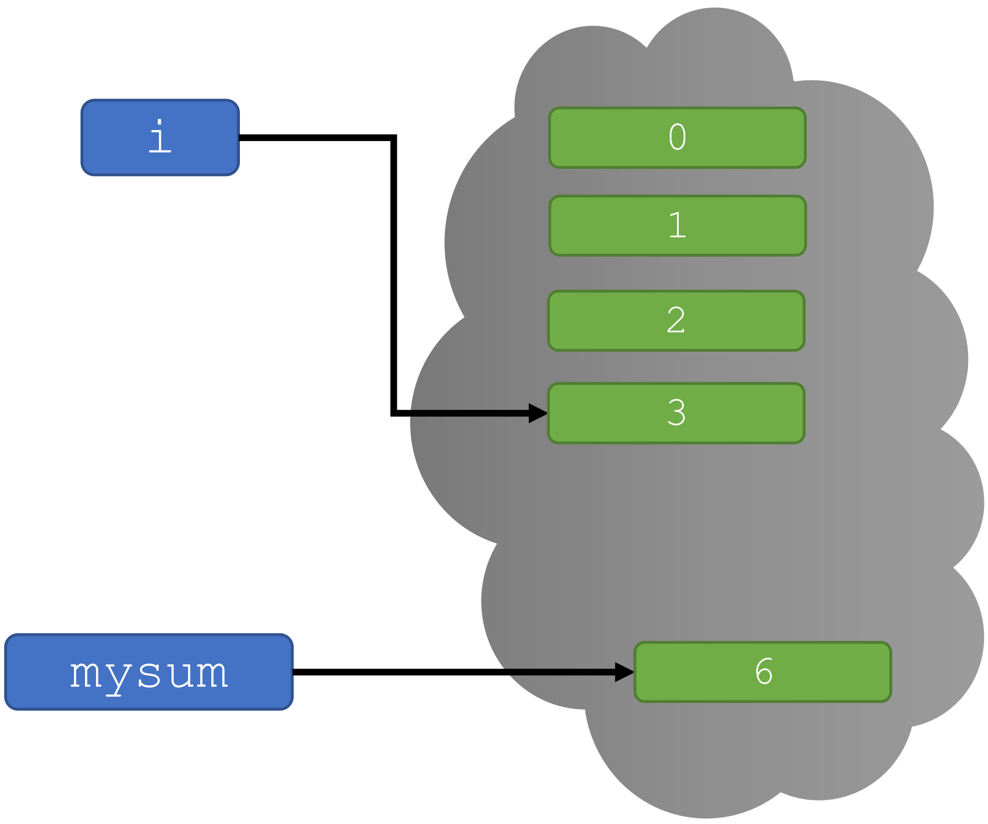
```
mysum = 0
for i in range(10):
    mysum += i
print(mysum)
```



RUNNING SUM

- `mysum` is a variable to store the **running sum**
- `range(10)` makes `i` be 0 then 1 then 2 then … then 9

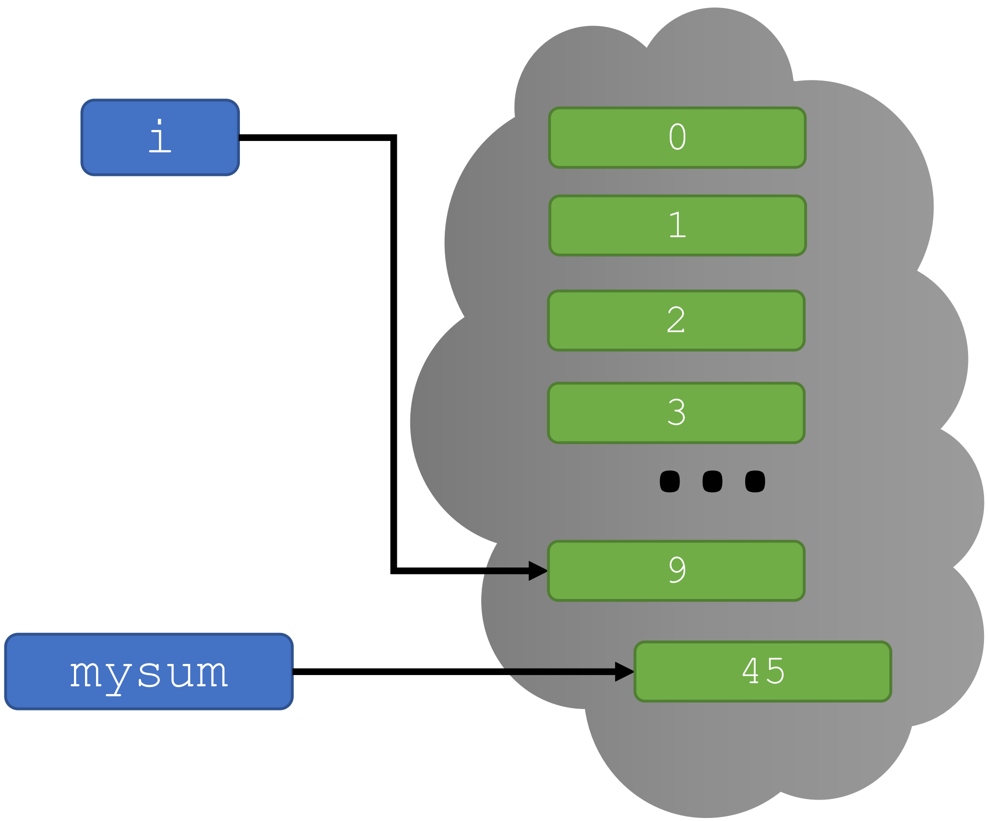
```
mysum = 0
for i in range(10):
    mysum += i
print(mysum)
```



RUNNING SUM

- `mysum` is a variable to store the **running sum**
- `range(10)` makes `i` be 0 then 1 then 2 then … then 9

```
mysum = 0
for i in range(10):
    mysum += i
print(mysum)
```



YOU TRY IT!

- Fix this code to use variables `start` and `end` in the `range`, to get the total sum between and including those values.
- For example, if `start=3` and `end=5` then the sum should be 12.

```
mysum = 0
start = ??
end = ??
for i in range(start, end):
    mysum += i
print(mysum)
```

for LOOPS and range

- Factorial implemented with a `while` loop (seen this already) and a `for` loop

```
x = 4
i = 1
factorial = 1
while i <= x:
    factorial *= i
    i += 1
print(f'{x} factorial is {factorial}')
```

Uses a while loop

```
x = 4
factorial = 1
for i in range(1, x+1, 1):
    factorial *= i
print(f'{x} factorial is {factorial}')
```

Uses a for loop

BIG IDEA

`for` loops only repeat for however long the sequence is

The loop variables takes on these values in order.

SUMMARY

- Looping mechanisms
  - `while` and `for` loops
  - Lots of **syntax** today, be sure to get lots of **practice**!
- While loops
  - Loop as long as a **condition is true**
  - Need to make sure you don't enter an **infinite loop**
- For loops
  - Can loop over **ranges** of numbers
  - Can loop over **elements** of a string
  - Will soon see many other things are easy to loop over

LOOPS OVER STRINGS, GUESS-and-CHECK, BINARY

(download slides and .py files to follow along)

6.100L Lecture 4

Ana Bell

LAST TIME

- Looping mechanisms
  - `while` and `for` loops
- While loops
  - Loop as long as a **condition is true**
  - Need to make sure you don't enter an **infinite loop**
- For loops
  - Loop variable takes on values in a sequence, one at a time
  - Can loop over **ranges** of numbers
  - Will soon see many other things are easy to loop over

`break` STATEMENT

- Immediately exits whatever loop it is in
- Skips remaining expressions in code block
- **Exits only innermost loop**!

```
while <condition_1>:
    while <condition_2>:
        <expression_a>
        break
        <expression_b>
    <expression_c>
```

Evaluated when <condition_1> and <condition_2> are True

Never evaluated (don't write code like this)

Evaluated when <condition_1> is True

`break` STATEMENT

```
mysum = 0
for i in range(5, 11, 2):
    mysum += i
    if mysum == 5:
        break
        mysum += 1
print(mysum)
```

- What happens in this program?
- Python Tutor LINK

YOU TRY IT!

- Write code that loops a `for` loop over some range and prints how many even numbers are in that range. Try it with:
    - `range(5)`
    - `range(10)`
    - `range(2,9,3)`
    - `range(-4,6,2)`
    - `range(5,6)`

STRINGS and LOOPS

- Code to check for letter i or u in a string.
- All 3 do the same thing

```
s = "demo loops - fruit loops"
for index in range(len(s)):
    if s[index] == 'i' or s[index] == 'u':
        print("There is an i or u")
```

Uses range to iterate through index of s

```
for char in s:
    if char == 'i' or char == 'u':
        print("There is an i or u")
```

Iterates through characters of s directly

```
for char in s:
    if char in 'iu':
        print("There is an i or u")
```

Iterates through characters of s directly (most "pythonic")

BIG IDEA

The sequence of values in a `for` loop isn't limited to numbers

ROBOT CHEERLEADERS

```
an_letters = "aefhilmnorsxAEFHILMNORSX"

word = input("I will cheer for you! Enter a word: ")
times = int(input("Enthusiasm level (1-10): "))

for c in word:
    if c in an_letters:
        print(f'Give me an {c}: {c}')
    else:
        print(f'Give me a {c}: {c}')
print("What's that spell?")
for i in range(times):
    print(word, '!!!')
```

c is a loop variable whose value is each letter that the user gave

i is a loop variable whose value is 0 through times-1, one at a time

YOU TRY IT!

- Assume you are given a string of lowercase letters in variable s. Count how many unique letters there are in the string. For example, if

```
s = "abca"
```

Then your code prints 3.

HINT:
Go through each character in s.
Keep track of ones you've seen in a string variable.
Add characters from s to the seen string variable if they are not already a character in that seen variable.

SUMMARY SO FAR

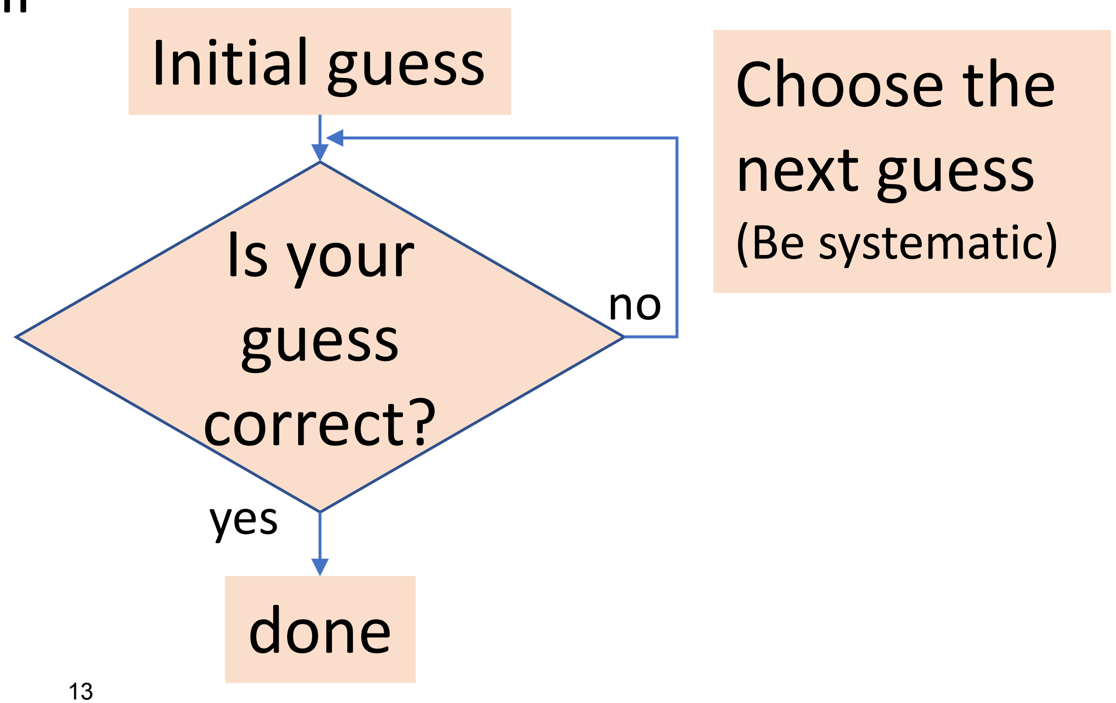
- Objects have **types**
- Expressions are **evaluated to one value**, and bound to a variable name
- Branching
  - if, else, elif
  - Program executes **one set of code or another**
- Looping mechanisms
  - `while` and `for` loops
  - Code executes repeatedly **while some condition is true**
  - Code executes repeatedly **for all values in a sequence**

THAT IS ALL YOU NEED TO IMPLEMENT ALGORITHMS

GUESS-and-CHECK

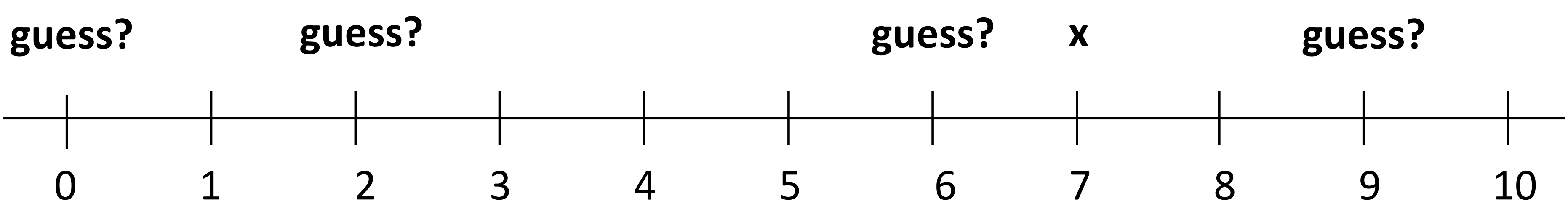
GUESS-and-CHECK

- Process called **exhaustive enumeration**
- Applies to a problem where …
  - You are able to **guess a value** for solution
  - You are able to **check if the solution is correct**
- You can **keep guessing** until
  - Find solution or
  - Have guessed all values



GUESS-and-CHECK SQUARE ROOT

- Basic idea:
  - Given an `int`, call it `x`, want to see if there is another `int` which is its square root
  - Start with a `guess` and check if it is the right answer

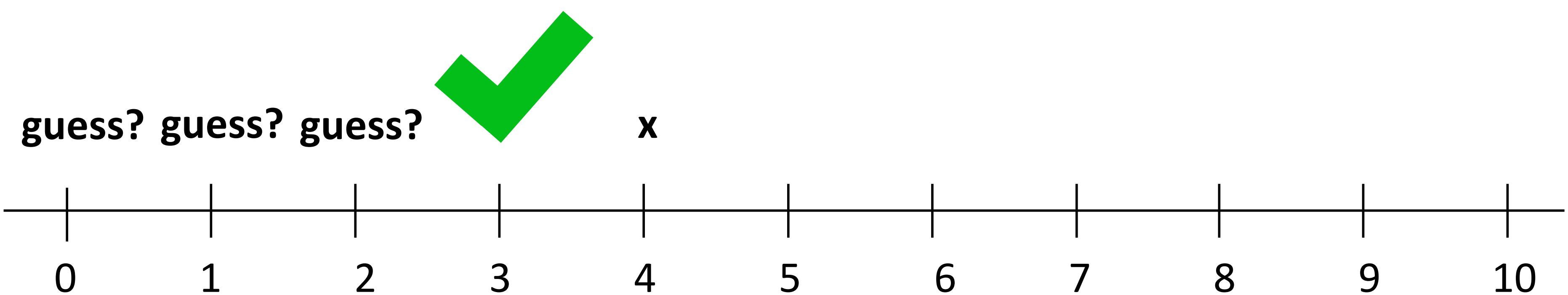


GUESS-and-CHECK SQUARE ROOT

- Basic idea:
  - Given an `int`, call it `x`, want to see if there is another `int` which is its square root
  - Start with a `guess` and check if it is the right answer
  - To be **systematic**, start with `guess` = 0, then 1, then 2, etc

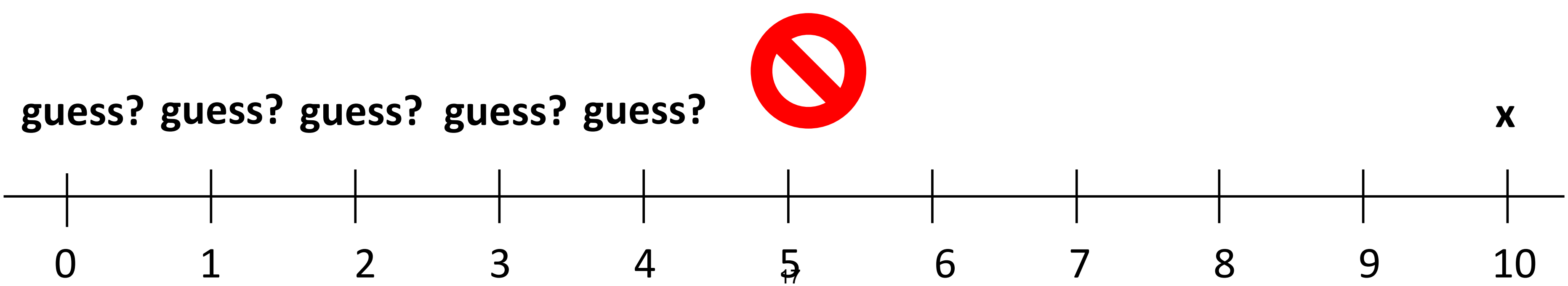
GUESS-and-CHECK SQUARE ROOT

- Basic idea:
  - Given an `int`, call it `x`, want to see if there is another `int` which is its square root
  - Start with a `guess` and check if it is the right answer
  - To be **systematic**, start with `guess` = 0, then 1, then 2, etc
- If `x` is a **perfect square**, we will **eventually find its root** and can stop (look at guess squared)



GUESS-and-CHECK SQUARE ROOT

- Basic idea:
  - Given an `int`, call it `x`, want to see if there is another `int` which is its square root
  - Start with a `guess` and check if it is the right answer
  - To be **systematic**, start with `guess` = 0, then 1, then 2, etc
- But what if `x` is **not a perfect square**?
  - Need to know when to stop
  - **Use algebra** – if `guess` squared is bigger than `x`, then can stop



GUESS-and-CHECK SQUARE ROOT with while loop

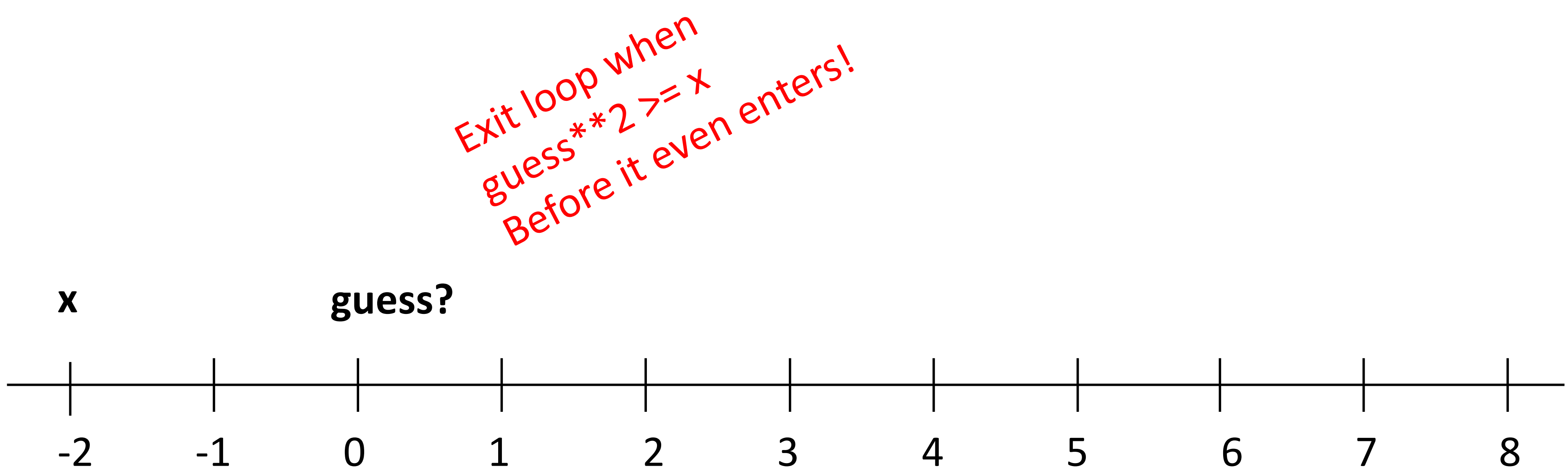
```
guess = 0
x = int(input("Enter an integer: "))
while guess**2 < x:
    guess = guess + 1
if guess**2 == x:
    print("Square root of", x, "is", guess)
else:
    print(x, "is not a perfect square")
```

Exit loop when guess**2 >= x

Check why you exited the loop

GUESS-and-CHECK SQUARE ROOT

- Does this work for any integer value of `x`?
- What if `x` is negative?
  - `while` loop immediately terminates
- Could **check for negative input**, and handle differently



GUESS-and-CHECK SQUARE ROOT with while loop

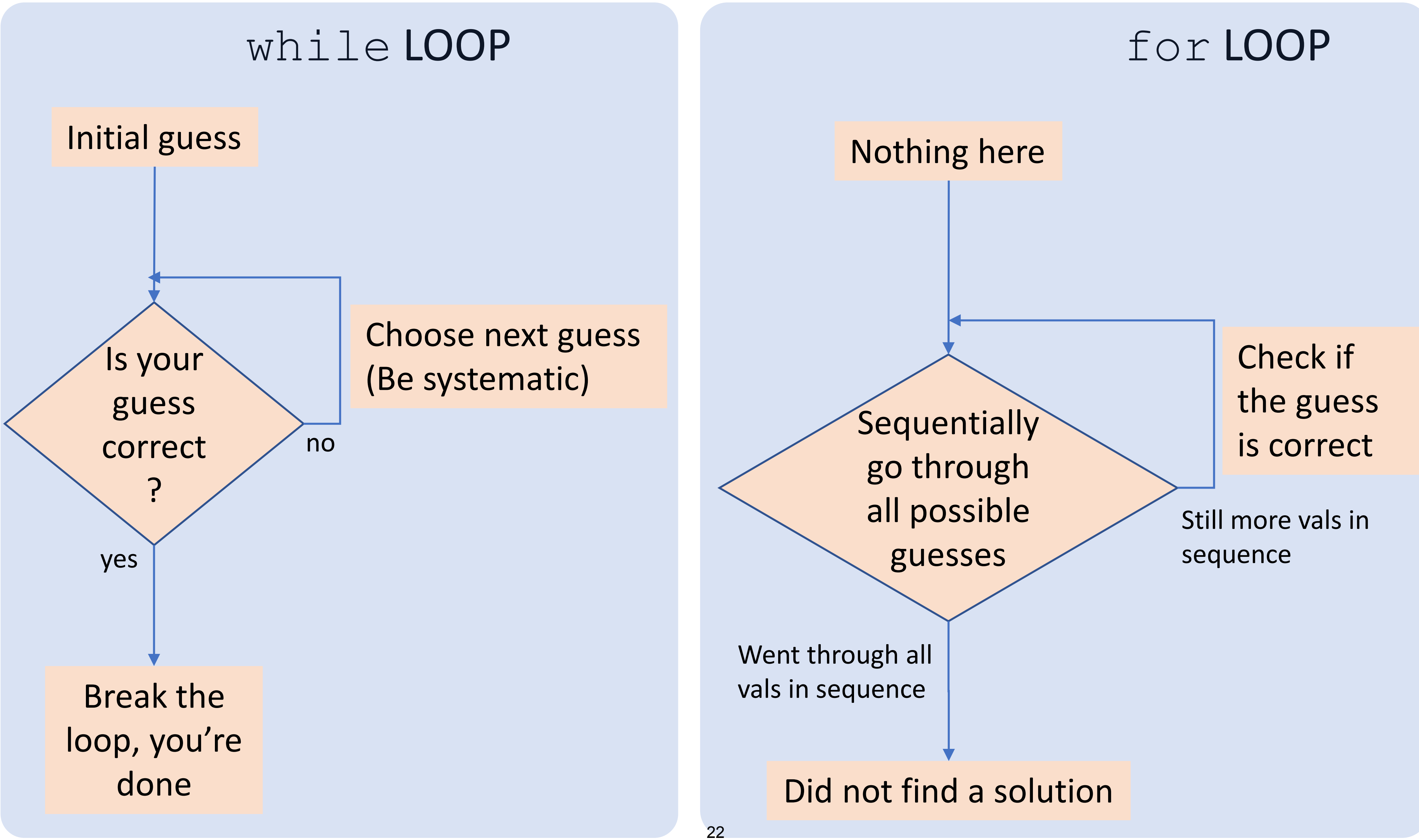
```
guess = 0
neg_flag = False
x = int(input("Enter a positive integer: "))
if x < 0:
    neg_flag = True
while guess**2 < x:
    guess = guess + 1
if guess**2 == x:
    print("Square root of", x, "is", guess)
else:
    print(x, "is not a perfect square")
    if neg_flag:
        print("Just checking... did you mean", -x, "?")
```

BIG IDEA

Guess-and-check can't test an infinite number of values

You have to stop at some point!

GUESS-and-CHECK COMPARED



YOU TRY IT!

- Hardcode a number as a secret number.
- Write a program that checks through all the numbers from 1 to 10 and prints the secret value if it's in that range. **If it's not found, it doesn't print anything.**
- How does the program look if I change the requirement to be: **If it's not found, prints that it didn't find it.**

YOU TRY IT!

- Compare the two codes that:
  - Hardcode a number as a secret number.
  - Checks through all the numbers from 1 to 10 and prints the secret value if it's in that range.

If it's not found, it **doesn't print anything**.

Answer:

```
secret = 7

for i in range(1,11):
    if i == secret:
        print("yes, it's", i)
```

If it's not found, **prints that it didn't find it**.

Answer:

```
secret = 7
found = False
for i in range(1,11):
    if i == secret:
        print("yes, it's", i)
        found = True
if not found:
    print("not found")
```

BIG IDEA

Booleans can be used as signals that something happened

We call them Boolean flags.

`while` LOOP or `for` LOOP?

- Already saw that code looks **cleaner when iterating over sequences** of values (i.e. using a `for` loop)
  - Don’t set up the iterant yourself as with a while loop
  - Less likely to introduce errors
- Consider an example that uses a `for` loop and an explicit `range` of values

GUESS-and-CHECK CUBE ROOT: POSITIVE CUBES

```
cube = int(input("Enter an integer: "))

for guess in range(cube+1):
    if guess**3 == cube:
        print("Cube root of", cube, "is", guess)
```

Want to include cube when cube is 1

GUESS-and-CHECK CUBE ROOT: POSITIVE and NEGATIVE CUBES

```
cube = int(input("Enter an integer: "))

for guess in range(abs(cube)+1):
    if guess**3 == abs(cube):
        if cube < 0:
            guess = -guess
        print("Cube root of "+str(cube)+" is "+str(guess))
```

Assume it's positive

Deal with negative cube here

GUESS-and-CHECK CUBE ROOT: JUST a LITTLE FASTER

```
cube = int(input("Enter an integer: "))
for guess in range(abs(cube)+1):
    if guess**3 >= abs(cube):
        break
if guess**3 != abs(cube):
    print(cube, "is not a perfect cube")
else:
    if cube < 0:
        guess = -guess
    print("Cube root of "+str(cube)+" is "+str(guess))
```

Terminate search once know you have passed possible answer

Check why you exited the loop and decide if the guess is not a perfect cube

ANOTHER EXAMPLE

- Remember those word problems from your childhood?
- For example:
  - Alyssa, Ben, and Cindy are selling tickets to a fundraiser
  - Ben sells 2 fewer than Alyssa
  - Cindy sells twice as many as Alyssa
  - 10 total tickets were sold by the three people
  - How many did Alyssa sell?
- Could solve this algebraically, but we can also use guess-and-check

GUESS-and-CHECK with WORD PROBLEMS

Check all possible values

For each value of alyssa, check all possible values

For each pair of alyssa and ben, check all possible values

```
for alyssa in range(11):
    for ben in range(11):
        for cindy in range(11):
            total = (alyssa + ben + cindy == 10)
            two_less = (ben == alyssa-2)
            twice = (cindy == 2*alyssa)
            if total and two_less and twice:
                print(f"Alyssa sold {alyssa} tickets")
                print(f"Ben sold {ben} tickets")
                print(f"Cindy sold {cindy} tickets")
```

3 Booleans for our word problem equations

Solution found when all 3 hold

EXAMPLE WITH BIGGER NUMBERS

- With bigger numbers, nesting loops is slow!
- For example:
  - Alyssa, Ben, and Cindy are selling tickets to a fundraiser
  - Ben sells **20** fewer than Alyssa
  - Cindy sells **twice** as many as Alyssa
  - **1000** total tickets were sold by the three people
  - How many did Alyssa sell?
  - The previous code won’t end in a reasonable time
- Instead, loop over one variable and code the equations directly

MORE EFFICIENT SOLUTION

One loop over one variable

Replace loops with direct calculation for other 2 values/people

Last condition

```
for alyssa in range(1001):
    ben = max(alyssa - 20, 0)
    cindy = alyssa * 2
    if ben + cindy + alyssa == 1000:
        print("Alyssa sold " + str(alyssa) + " tickets")
        print("Ben sold " + str(ben) + " tickets")
        print("Cindy sold " + str(cindy) + " tickets")
```

BIG IDEA

You can apply computation to many problems!

BINARY NUMBERS

NUMBERS in PYTHON

- **int**
  - integers, like the ones you learned about in elementary school
- **float**
  - reals, like the ones you learned about in middle school

OUR MOTIVATION - keep this in mind for the next few slides

```
x = 0
for i in range(10):
    x += 0.1
print(x == 1)
print(x, '==', 10*0.1)
```

Note: x += 0.1 is the same as x = x + 0.1

0.9999999999999999 == 1.0

BIG IDEA

Operations on some floats introduces a very small error.

The small error can have a big effect if operations are done many times!

A CLOSER LOOK AT FLOATS

- Python (and every other programming language) uses “floating point” to **approximate real numbers**
- The term “floating point” refers to the way these numbers are stored in computer
- Approximation usually doesn’t matter
  - But it does for us!
  - Let’s see why…

FLOATING POINT REPRESENTATION

- Depends on computer hardware, not programming language implementation
- Key things to understand
  - Numbers (and everything else) are represented as a **sequence of bits** (0 or 1).
  - When **we** write numbers down, the notation uses base 10.
    - 0.1 stands for the rational number 1/10
  - This produces **cognitive dissonance** – and it will influence how we write code

WHY BINARY?
HARDWARE IMPLEMENTATION

- Easy to implement in hardware—build components that can be in **one** of **two** states
- Computer hardware is built around methods that can efficiently store information as 0's or 1's and do arithmetic with this rep
  - a voltage is "high" or "low" a magnetic spin is "up" or "down"
- Fine for integer arithmetic, but what about numbers with fractional parts (floats)?

BINARY NUMBERS

- Base 10 representation of an integer
  - sum of powers of 10, scaled by integers from 0 to 9

$1507 = 1*10^3 + 5*10^2 + 0*10^1 + 7*10^0$

$= 1000 + 500 + 7$

- Binary representation is same idea in base 2
  - sum of powers of 2, scaled by integers from 0 to 1
- $1507_{10} = 1*2^{10} + 1*2^8 + 1*2^7 + 1*2^6 + 1*2^5 + 1*2^1 + 1*2^0$

$= \boxed{1024} + 256 + 128 + 64 + 32 + 2 + 1$

$= 2^{10} + 2^8 + 2^7 + 2^6 + 2^5 + 2^1 + 2^0$

$= 10111100011_2$

Highest power of 2 to get us closest without going over to 1507

CONVERTING DECIMAL INTEGER TO BINARY

- We input integers in decimal, computer needs to convert to binary
- Consider example of
  - $x = 19_{10} = 1*2^4 + 0*2^3 + 0*2^2 + 1*2^1 + 1*2^0 = \boxed{1001}1$
- If we take **remainder of x relative to 2** `(x%2)`, that gives us the last binary bit
- If we then **integer divide x by 2** `(x//2)`, all the bits get shifted right
  - $x//2 = 1*2^3 + 0*2^2 + 0*2^1 + 1*2^0 = 1001$
- Keep doing **successive divisions**; now remainder gets next bit, and so on
- Let's convert to binary form

DOING THIS in PYTHON for POSITIVE NUMBERS

[Python Tutor LINK]

```
result = ''
if num == 0:
    result = '0'
while num > 0:
    result = str(num%2) + result
    num = num//2
```

DOING this in PYTHON and HANDLING NEGATIVE NUMBERS

```
if num < 0:
    is_neg = True
    num = abs(num)
else:
    is_neg = False
result = ''
if num == 0:
    result = '0'
while num > 0:
    result = str(num%2) + result
    num = num//2
if is_neg:
    result = '-' + result
```

Set a negative flag and handle it

SUMMARY

- Loops can iterate over any sequence of values:
  - range for numbers
  - A string
- Guess-and-check provides a **simple algorithm** for solving problems
  - When set of **potential solutions is enumerable**, exhaustive enumeration guaranteed to work (eventually)
- Binary numbers help us understand how the machine works
  - Converting to binary will help us understand how decimal numbers are stored
  - Important for the next algorithm we will see

FLOATS and APPROXIMATION METHODS

(download slides and .py files to follow along)

6.100L Lecture 5

Ana Bell

OUR MOTIVATION FROM LAST LECTURE

```
x = 0
for i in range(10):
    x += 0.1
print(x == 1)
print(x, '==', 10*0.1)
```

0.9999999999999999 == 1.0

INTEGERS

- Integers have straightforward representations in binary
- The code was simple (and can add a piece to deal with negative numbers)

```
if num < 0:
    is_neg = True
    num = abs(num)
else:
    is_neg = False
result = ''
if num == 0:
    result = '0'
while num > 0:
    result = str(num%2) + result
    num = num//2
if is_neg:
    result = '-' + result
```

Set a negative flag and handle it

FRACTIONS

FRACTIONS

- What does the decimal fraction 0.abc mean?
  - $a*10^{-1} + b*10^{-2} + c*10^{-3}$
- For binary representation, we use the same idea
  - $a*2^{-1} + b*2^{-2} + c*2^{-3}$
- Or to put this in simpler terms, the binary representation of a decimal fraction f would require finding the values of a, b, c, etc. such that
  - $f = 0.5a + 0.25b + 0.125c + 0.0625d + 0.03125e + \ldots$

WHAT ABOUT FRACTIONS?

- How might we find that representation?
- In decimal form: $3/8 = 0.\boxed{3}\boxed{7}\boxed{5} = \boxed{3*10^{-1}} + \boxed{7*10^{-2}} + \boxed{5*10^{-3}}$

- **Recipe idea**: if we can multiply by a power of 2 big enough to turn into a whole number, can convert to binary, and then divide by the same power of 2 to restore
  - $0.375 * (2**3) = 3_{10}$
  - Convert 3 to binary (now 11_2)
  - Divide by 2**3 (shift right three spots) to get 0.011_2

BUT…

- If there is **no integer p such that x*(2^p) is a whole number**, then internal representation is **always** an approximation
- And I am assuming that the representation for the decimal fraction I provided as input is completely accurate and not already an approximation as a result of number being read into Python
- Floating point conversion works:
  - Precisely for numbers like 3/8
  - But not for 1/10
  - **One has a power of 2 that converts to whole number, the other doesn't**

TRACE THROUGH THIS ON YOUR OWN
Python Tutor LINK

% grabs the decimal part only
e.g. 1.1%1 gives 0.1

```
x =0.625
```

Find power of 2 to make integer

```
p = 0
while ((2**p)*x)%1 != 0:
    print('Remainder = ' + str((2**p)*x - int((2**p)*x)))
    p += 1
```

Convert to int

```
num = int(x*(2**p))
```

Encode as binary number, same as prev slide

```
result = ''
if num == 0:
    result = '0'
while num > 0:
    result = str(num%2) + result
    num = num//2
```

Pad front with 0's, i.e. shift right

```
for i in range(p - len(result)):
    result = '0' + result
```

Insert decimal

```
result = result[0:-p] + '.' + result[-p:]
```

```
print('The binary representation of the decimal ' + str(x) + ' is ' + str(result))
```

WHY is this a PROBLEM?

- What does the decimal representation 0.125 mean
  - $1*10^{-1} + 2*10^{-2} + 5*10^{-3}$
- Suppose we want to represent it in binary?
  - $1*2^{-3}$ 0.001
- How how about the decimal representation 0.1
  - In base 10: $1 * 10^{-1}$
  - In base 2: ?

0.00011001100110011001100110011...
Infinite!

THE POINT?

- If **everything ultimately is represented in terms of bits**, we need to think about how to use binary representation to capture numbers
- Integers are straightforward
- But real numbers (things with digits after the decimal point) are a problem
  - The idea was to try and convert a real number to an int by multiplying the real with some multiple of 2 to get an int
  - Sometimes there is no such power of 2!
  - Have to somehow **approximate the potentially infinite binary sequence** of bits needed to represent them

FLOATS

STORING FLOATING POINT NUMBERS #.#

- Floating point is a pair of integers
  - Significant digits and base 2 exponent
  - (1, 1) → $1*2^{1}$ → 10_2 → 2.0
  - (1, -1) → $1*2^{-1}$ → 0.1_2 → 0.5
  - (125, -2) → $125*2^{-2}$ → 11111.01_2 → 31.25

  125 is 1111101 then move the decimal point over 2

Called "floating point" because location of decimal can "float" relative to significant digits

USE A FINITE SET OF BITS TO REPRESENT A POTENTIALLY INFINITE SET OF BITS

- The maximum number of significant digits governs the precision with which numbers can be represented
- Most modern computers use **32 bits** to represent significant digits
- If a number is represented with more than 32 bits in binary, the **number will be rounded**
  - Error will be at the 32nd bit
  - **Error will only be on order of $2*10^{-10}$**

2^{-32} is approx. 10^{-10}
pretty small number, isn't it?

SURPRISING RESULTS!

```
x = 0
for i in range(10):
    x += 0.125
print(x == 1.25)
```

True

```
x = 0
for i in range(10):
    x += 0.1
print(x == 1)
```

False

```
print(x, '==', 10*0.1)
```

0.9999999999999999 == 1.0

MORAL of the STORY

- **Never** use == to test floats
  - Instead test whether they are within small amount of each other
- What gets **printed** isn't always what is in **memory**
- Need to be **careful** in designing algorithms that use floats

APPROXIMATION METHODS

LAST LECTURE

- Guess-and-check provides a **simple algorithm** for solving problems
- When set of **potential solutions is enumerable**, exhaustive enumeration guaranteed to work (eventually)
- It’s a limiting way to solve problems
  - Increment is **usually an integer but not always**. i.e. we just need some pattern to give us a finite set of enumerable values
  - Can’t give us an approximate solution to varying degrees

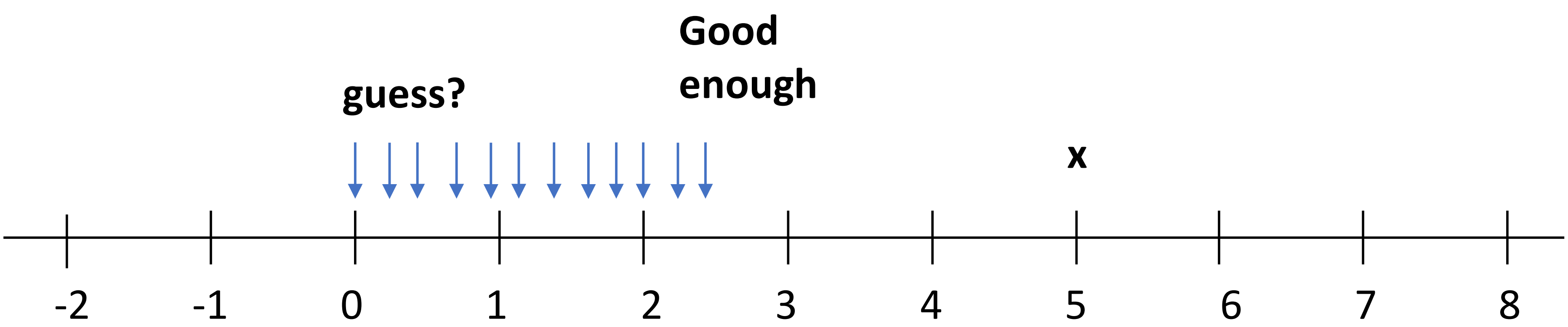
BETTER than GUESS-and-CHECK

- Want to find an **approximation to an answer**
  - Not just the correct answer, like guess-and-check
  - And not just that we did not find the answer, like guess-and-check

EFFECT of APPROXIMATION on our ALGORITHMS?

- **Exact** answer may not be **accessible**
- Need to find ways to get **"good enough" answer**
  - Our answer is "close enough" to ideal answer
- Need ways to deal with fact that exhaustive enumeration can't test every possible value, since set of possible answers is in principle infinite
- Floating point **approximation errors** are important to this method
  - Can't rely on equality!

APPROXIMATE sqrt(x)



FINDING ROOTS

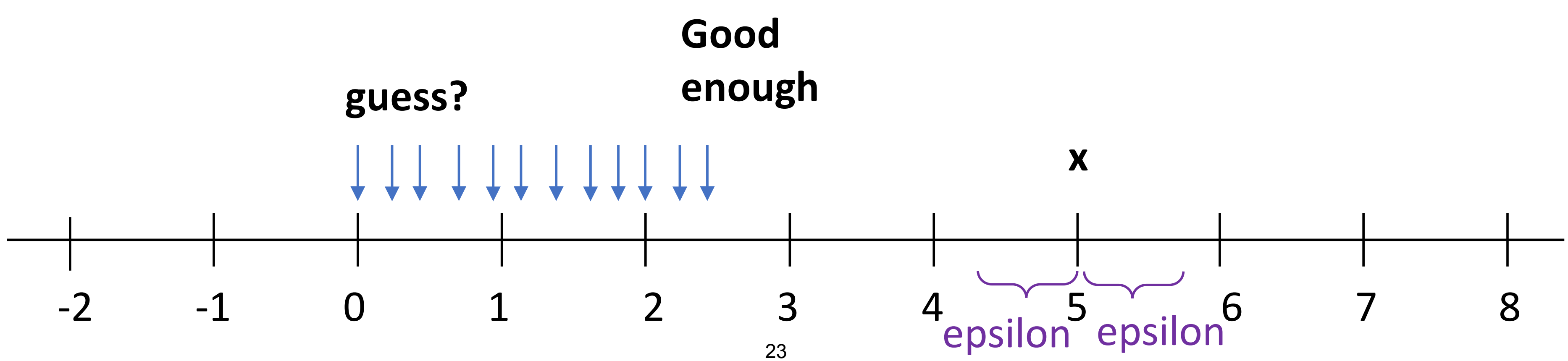
- Last lecture we looked at using exhaustive enumeration/guess and check methods to find the **roots of perfect squares**
- Suppose we want to find the square root of any positive integer, or any positive number
- Question: What does it mean to find the square root of x?
  - Find an r such that r*r = x ?
  - If x is not a perfect square, then not possible in general to find an exact r that satisfies this relationship; and **exhaustive search is infinite**

APPROXIMATION

- Find an answer that is **"good enough"**
  - E.g., find a r such that r*r is within a given (small) distance of x
  - Use epsilon: given x we want to find r such that $|r^2\text{-x}|<\varepsilon$
- Algorithm
  - Start with guess **known to be too small** – call it `g`
  - Increment by a small value – call it `a` – to give a new guess `g`
  - Check if `g**2` is close enough to `x` (within ε)
  - Continue until get answer close enough to actual answer
- Looking at all possible **values `g + k*a`** for integer values of **`k`** – so similar to exhaustive enumeration
  - But cannot test all possibilities as infinite

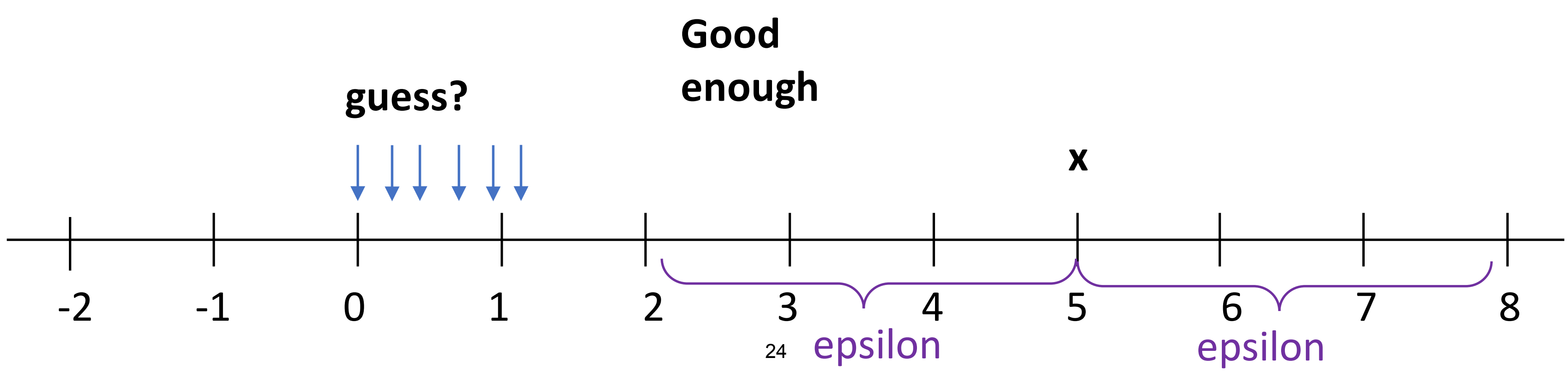
APPROXIMATION ALGORITHM

- In this case, we have **two parameters** to set
  - **epsilon** (how close are we to answer?)
  - **increment** (how much to increase our guess?)
- Performance will vary based on these values
  - In speed
  - In accuracy
- **Decreasing increment** size → slower program, but more likely to get good answer (and vice versa)



APPROXIMATION ALGORITHM

- In this case, we have **two parameters** to set
  - **epsilon** (how close are we to answer?)
  - **increment** (how much to increase our guess?)
- Performance will vary based on these values
  - In speed
  - In accuracy
- **Increasing epsilon** → less accurate answer, but faster program (and vice versa)



BIG IDEA

Approximation is like guess-and-check except…

1) We increment by some small amount
2) We stop when close enough (exact is not possible)

IMPLEMENTATION

```
x = 36
epsilon = 0.01
num_guesses = 0
guess = 0.0
increment = 0.0001
```

```
while abs(guess**2 - x) >= epsilon:
    guess += increment
    num_guesses += 1
```

```
print('num_guesses =', num_guesses)
print(guess, 'is close to square root of', x)
```

Will this loop always terminate?

Note: guess += increment is same as guess = guess + increment

OBSERVATIONS with DIFFERENT VALUES for x

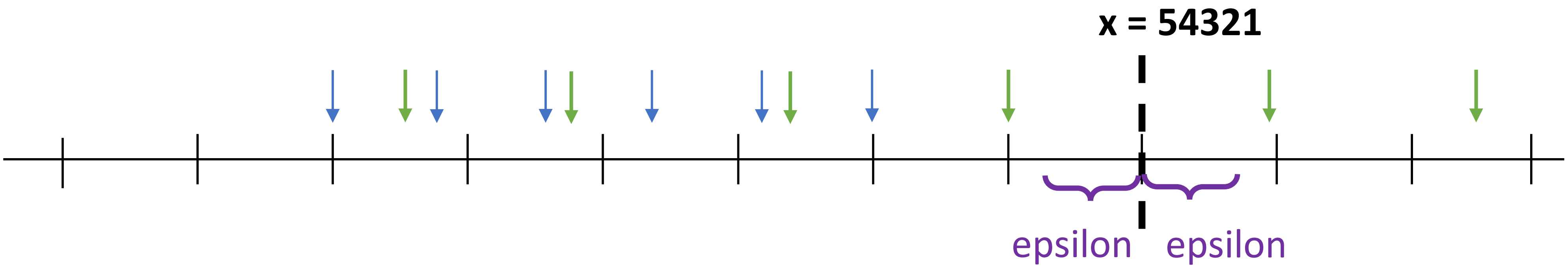
- For x = 36
  - Didn’t find 6
  - Took about 60,000 guesses
- Let’s try:
  - 24
  - 2
  - 12345
  - 54321

```python
x = 54321
epsilon = 0.01
numGuesses = 0
guess = 0.0
increment = 0.0001

while abs(guess**2 - x) >= epsilon:
    guess += increment
    numGuesses += 1
    if numGuesses%100000 == 0:
        print('Current guess =', guess)
        print('Current guess**2 - x =', abs(guess*guess - x))
print('numGuesses =', numGuesses)
print(guess, 'is close to square root of', x)
```

Debugging print statements every 100000 times through the loop, showing guess and how far away from epsilon we are

WE OVERSHOT the EPSILON!

- Blue arrow is the `guess`
- Green arrow is `guess**2`



SOME OBSERVATIONS

- Decrementing function eventually starts incrementing
  - So didn't exit loop as expected
- We have **over-shot the mark**
  - I.e., we jumped from a value too far away but too small to one too far away but too large
- We **didn't account for this possibility when writing the loop**
- Let's fix that

LET'S FIX IT

```
x = 54321
epsilon = 0.01
numGuesses = 0
guess = 0.0
increment = 0.0001
while abs(guess**2 - x) >= epsilon and guess**2 <= x:
    guess += increment
    numGuesses += 1
print('numGuesses =', numGuesses)
if abs(guess**2 - x) >= epsilon:
    print('Failed on square root of', x)
else:
    print(guess, 'is close to square root of', x)
```

Same condition as guess-and-check, stop when you go past the last reasonable guess

Exited b/c guess**2 > x

Exited b/c guess**2 is within eps

BIG IDEA

It’s possible to overshoot the epsilon, so you need another end condition

SOME OBSERVATIONS

- Now it stops, but **reports failure**, because it has over-shot the answer
- Let’s try resetting increment to 0.00001
  - Smaller increment means **more values will be checked**
  - Program will be slower

BIG IDEA

Be careful when comparing floats.

LESSONS LEARNED in APPROXIMATION

- Can't use == to check an exit condition
- Need to be careful that looping mechanism doesn't **jump over exit test** and loop forever
- **Tradeoff** exists between efficiency of algorithm and accuracy of result
- Need to think about **how close** an answer we want when **setting parameters** of algorithm
- To get a good answer, this method can be painfully slow.
  - Is there a **faster way that still gets good answers**?
  - **YES!** We will see it next lecture....

SUMMARY

- Floating point numbers introduce challenges!
- They **can't be represented in memory exactly**
  - Operations on floats introduce tiny errors
  - Multiple operations on floats **magnify errors** :(
- Approximation methods use floats
  - Like guess-and-check except that
    (1) We use a float as an **increment**
    (2) We stop when we are **close enough**
  - **Never use == to compare floats** in the stopping condition
  - Be careful about **overshooting** the close-enough stopping condition

FLOATS and APPROXIMATION METHODS

(download slides and .py files to follow along)

6.100L Lecture 5

Ana Bell

OUR MOTIVATION FROM LAST LECTURE

```
x = 0
for i in range(10):
    x += 0.1
print(x == 1)
print(x, '==', 10*0.1)
```

0.9999999999999999 == 1.0

INTEGERS

- Integers have straightforward representations in binary
- The code was simple (and can add a piece to deal with negative numbers)

```
if num < 0:
    is_neg = True
    num = abs(num)
else:
    is_neg = False
result = ''
if num == 0:
    result = '0'
while num > 0:
    result = str(num%2) + result
    num = num//2
if is_neg:
    result = '-' + result
```

Set a negative flag and handle it

FRACTIONS

FRACTIONS

- What does the decimal fraction 0.abc mean?
  - $a*10^{-1} + b*10^{-2} + c*10^{-3}$
- For binary representation, we use the same idea
  - $a*2^{-1} + b*2^{-2} + c*2^{-3}$
- Or to put this in simpler terms, the binary representation of a decimal fraction f would require finding the values of a, b, c, etc. such that
  - f = 0.5a + 0.25b + 0.125c + 0.0625d + 0.03125e + …

WHAT ABOUT FRACTIONS?

- How might we find that representation?
- In decimal form: $3/8 = 0.\boxed{3}\boxed{7}\boxed{5} = \boxed{3*10^{-1}} + \boxed{7*10^{-2}} + \boxed{5*10^{-3}}$

- **Recipe idea**: if we can multiply by a power of 2 big enough to turn into a whole number, can convert to binary, and then divide by the same power of 2 to restore
  - $0.375 * (2**3) = 3_{10}$
  - Convert 3 to binary (now 11_2)
  - Divide by 2**3 (shift right three spots) to get 0.011_2

BUT…

- If there is **no integer p such that $x*(2^p)$ is a whole number**, then internal representation is **always** an approximation
- And I am assuming that the representation for the decimal fraction I provided as input is completely accurate and not already an approximation as a result of number being read into Python
- Floating point conversion works:
  - Precisely for numbers like 3/8
  - But not for 1/10
  - **One has a power of 2 that converts to whole number, the other doesn't**

TRACE THROUGH THIS ON YOUR OWN
Python Tutor LINK

% grabs the decimal part only
e.g. 1.1%1 gives 0.1

```python
x =0.625
```

Find power of 2 to make integer

```python
p = 0
while ((2**p)*x)%1 != 0:
    print('Remainder = ' + str((2**p)*x - int((2**p)*x)))
    p += 1
```

Convert to int

```python
num = int(x*(2**p))
```

Encode as binary number, same as prev slide

```python
result = ''
if num == 0:
    result = '0'
while num > 0:
    result = str(num%2) + result
    num = num//2
```

Pad front with 0's, i.e. shift right

```python
for i in range(p - len(result)):
    result = '0' + result
```

Insert decimal

```python
result = result[0:-p] + '.' + result[-p:]
```

```python
print('The binary representation of the decimal ' + str(x) + ' is ' + str(result))
```

WHY is this a PROBLEM?

- What does the decimal representation 0.125 mean
  - $1*10^{-1} + 2*10^{-2} + 5*10^{-3}$
- Suppose we want to represent it in binary?
  - $1*2^{-3}$ 0.001
- How how about the decimal representation 0.1
  - In base 10: $1 * 10^{-1}$
  - In base 2: ?

0.00011001100110011001100110011...

Infinite!

THE POINT?

- If **everything ultimately is represented in terms of bits**, we need to think about how to use binary representation to capture numbers
- Integers are straightforward
- But real numbers (things with digits after the decimal point) are a problem
  - The idea was to try and convert a real number to an int by multiplying the real with some multiple of 2 to get an int
  - Sometimes there is no such power of 2!
  - Have to somehow **approximate the potentially infinite binary sequence** of bits needed to represent them

FLOATS

STORING FLOATING POINT NUMBERS #.#

- Floating point is a pair of integers
  - Significant digits and base 2 exponent
  - $(1, 1) \rightarrow 1*2^{1} \rightarrow 10_2 \rightarrow 2.0$
  - $(1, -1) \rightarrow 1*2^{-1} \rightarrow 0.1_2 \rightarrow 0.5$
  - $(125, -2) \rightarrow 125*2^{-2} \rightarrow 11111.01_2 \rightarrow 31.25$

  125 is 1111101 then move the decimal point over 2

Called "floating point" because location of decimal can "float" relative to significant digits

USE A FINITE SET OF BITS TO REPRESENT A POTENTIALLY INFINITE SET OF BITS

- The maximum number of significant digits governs the precision with which numbers can be represented
- Most modern computers use **32 bits** to represent significant digits
- If a number is represented with more than 32 bits in binary, the **number will be rounded**
  - Error will be at the 32nd bit
  - **Error will only be on order of $2*10^{-10}$**

2^{-32} is approx. 10^{-10}
pretty small number, isn't it?

SURPRISING RESULTS!

```
x = 0
for i in range(10):
    x += 0.125
print(x == 1.25)
```

True

```
x = 0
for i in range(10):
    x += 0.1
print(x == 1)
```

False

```
print(x, '==', 10*0.1)
```

0.9999999999999999 == 1.0

MORAL of the STORY

- **Never** use == to test floats
  - Instead test whether they are within small amount of each other
- What gets **printed** isn't always what is in **memory**
- Need to be **careful** in designing algorithms that use floats

APPROXIMATION METHODS

LAST LECTURE

- Guess-and-check provides a **simple algorithm** for solving problems
- When set of **potential solutions is enumerable**, exhaustive enumeration guaranteed to work (eventually)
- It's a limiting way to solve problems
  - Increment is **usually an integer but not always**. i.e. we just need some pattern to give us a finite set of enumerable values
  - Can't give us an approximate solution to varying degrees

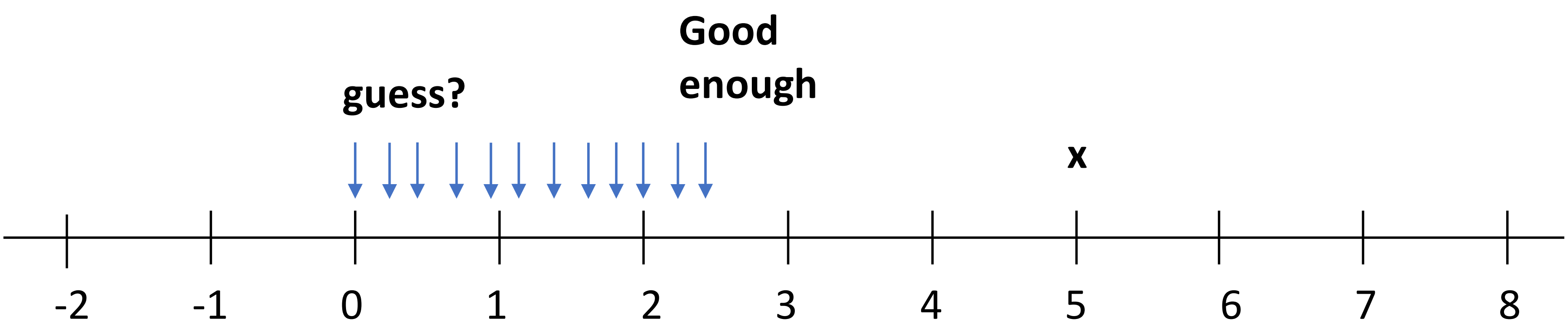
BETTER than GUESS-and-CHECK

- Want to find an **approximation to an answer**
  - Not just the correct answer, like guess-and-check
  - And not just that we did not find the answer, like guess-and-check

EFFECT of APPROXIMATION on our ALGORITHMS?

- **Exact** answer may not be **accessible**
- Need to find ways to get **"good enough" answer**
  - Our answer is "close enough" to ideal answer
- Need ways to deal with fact that exhaustive enumeration can't test every possible value, since set of possible answers is in principle infinite
- Floating point **approximation errors** are important to this method
  - Can't rely on equality!

APPROXIMATE sqrt(x)



FINDING ROOTS

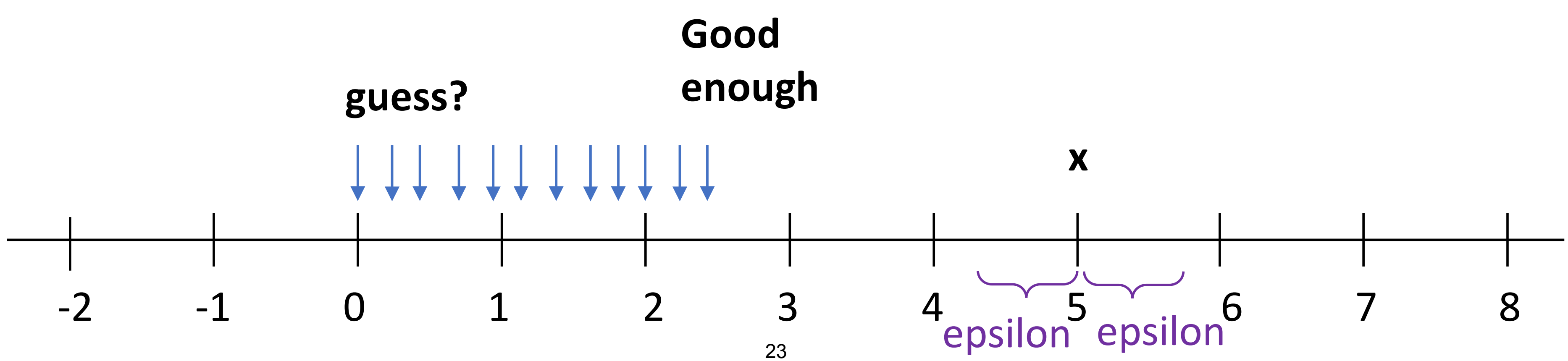
- Last lecture we looked at using exhaustive enumeration/guess and check methods to find the **roots of perfect squares**
- Suppose we want to find the square root of any positive integer, or any positive number
- Question: What does it mean to find the square root of x?
  - Find an r such that r*r = x ?
  - If x is not a perfect square, then not possible in general to find an exact r that satisfies this relationship; and **exhaustive search is infinite**

APPROXIMATION

- Find an answer that is **"good enough"**
  - E.g., find a r such that r*r is within a given (small) distance of x
  - Use epsilon: given x we want to find r such that $|r^2\text{-x}|<\varepsilon$
- Algorithm
  - Start with guess **known to be too small** – call it `g`
  - Increment by a small value – call it `a` – to give a new guess `g`
  - Check if `g**2` is close enough to `x` (within ε)
  - Continue until get answer close enough to actual answer
- Looking at all possible **values `g + k*a`** for integer values of **`k`** – so similar to exhaustive enumeration
  - But cannot test all possibilities as infinite

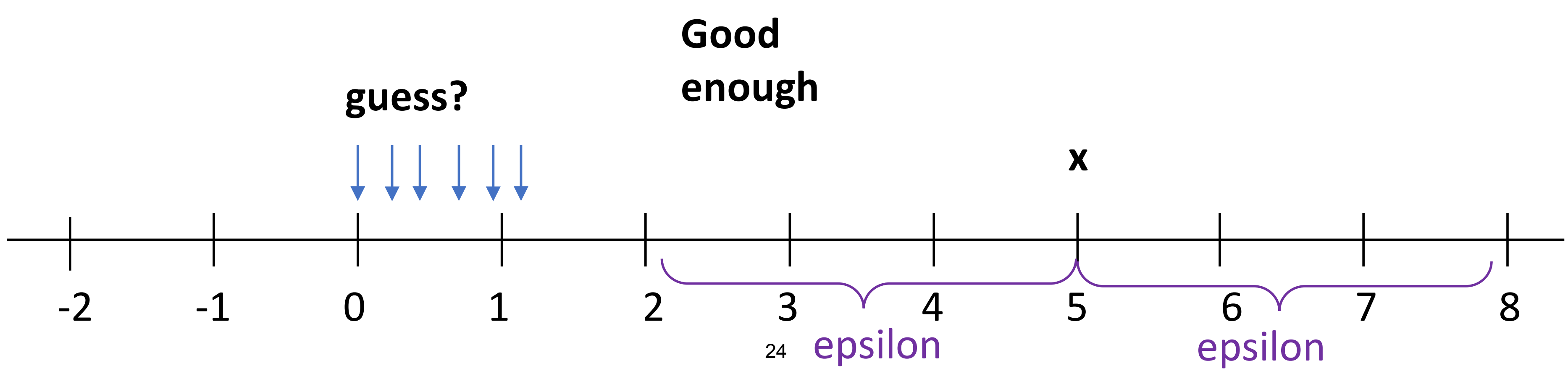
APPROXIMATION ALGORITHM

- In this case, we have **two parameters** to set
  - **epsilon** (how close are we to answer?)
  - **increment** (how much to increase our guess?)
- Performance will vary based on these values
  - In speed
  - In accuracy
- **Decreasing increment** size → slower program, but more likely to get good answer (and vice versa)



APPROXIMATION ALGORITHM

- In this case, we have **two parameters** to set
  - **epsilon** (how close are we to answer?)
  - **increment** (how much to increase our guess?)
- Performance will vary based on these values
  - In speed
  - In accuracy
- **Increasing epsilon** → less accurate answer, but faster program (and vice versa)



BIG IDEA

Approximation is like guess-and-check except…

1) We increment by some small amount
2) We stop when close enough (exact is not possible)

IMPLEMENTATION

```
x = 36
epsilon = 0.01
num_guesses = 0
guess = 0.0
increment = 0.0001
```

Will this loop always terminate?

```
while abs(guess**2 - x) >= epsilon:
    guess += increment
    num_guesses += 1
```

Note: guess += increment is same as guess = guess + increment

```
print('num_guesses =', num_guesses)
print(guess, 'is close to square root of', x)
```

OBSERVATIONS with DIFFERENT VALUES for x

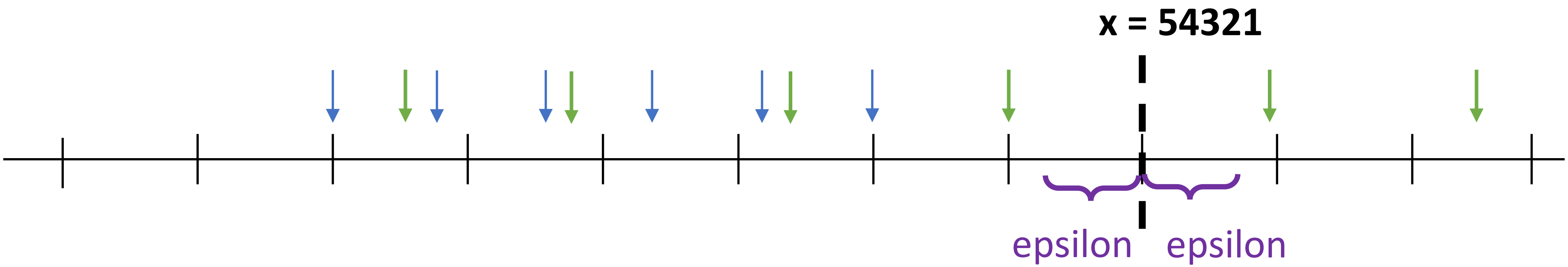
- For x = 36
  - Didn't find 6
  - Took about 60,000 guesses
- Let's try:
  - 24
  - 2
  - 12345
  - 54321

```python
x = 54321
epsilon = 0.01
numGuesses = 0
guess = 0.0
increment = 0.0001

while abs(guess**2 - x) >= epsilon:
    guess += increment
    numGuesses += 1
    if numGuesses%100000 == 0:
        print('Current guess =', guess)
        print('Current guess**2 - x =', abs(guess*guess - x))
print('numGuesses =', numGuesses)
print(guess, 'is close to square root of', x)
```

Debugging print statements every 100000 times through the loop, showing guess and how far away from epsilon we are

WE OVERSHOT the EPSILON!

- Blue arrow is the `guess`
- Green arrow is `guess**2`



SOME OBSERVATIONS

- Decrementing function eventually starts incrementing
  - So didn't exit loop as expected
- We have **over-shot the mark**
  - I.e., we jumped from a value too far away but too small to one too far away but too large
- We **didn't account for this possibility when writing the loop**
- Let's fix that

LET'S FIX IT

```
x = 54321
epsilon = 0.01
numGuesses = 0
guess = 0.0
increment = 0.0001
while abs(guess**2 - x) >= epsilon and guess**2 <= x:
    guess += increment
    numGuesses += 1
print('numGuesses =', numGuesses)
if abs(guess**2 - x) >= epsilon:
    print('Failed on square root of', x)
else:
    print(guess, 'is close to square root of', x)
```

Same condition as guess-and-check, stop when you go past the last reasonable guess

Exited b/c guess**2 > x

Exited b/c guess**2 is within eps

BIG IDEA

It’s possible to overshoot the epsilon, so you need another end condition

SOME OBSERVATIONS

- Now it stops, but **reports failure**, because it has over-shot the answer
- Let's try resetting increment to 0.00001
  - Smaller increment means **more values will be checked**
  - Program will be slower

BIG IDEA

Be careful when comparing floats.

LESSONS LEARNED in APPROXIMATION

- Can't use == to check an exit condition
- Need to be careful that looping mechanism doesn't **jump over exit test** and loop forever
- **Tradeoff** exists between efficiency of algorithm and accuracy of result
- Need to think about **how close** an answer we want when **setting parameters** of algorithm
- To get a good answer, this method can be painfully slow.
  - Is there a **faster way that still gets good answers**?
  - **YES!** We will see it next lecture....

SUMMARY

- Floating point numbers introduce challenges!
- They **can't be represented in memory exactly**
  - Operations on floats introduce tiny errors
  - Multiple operations on floats **magnify errors** :(
- Approximation methods use floats
  - Like guess-and-check except that
    (1) We use a float as an **increment**
    (2) We stop when we are **close enough**
  - **Never use == to compare floats** in the stopping condition
  - Be careful about **overshooting** the close-enough stopping condition

BISECTION SEARCH

(download slides and .py files to follow along)

6.100L Lecture 6

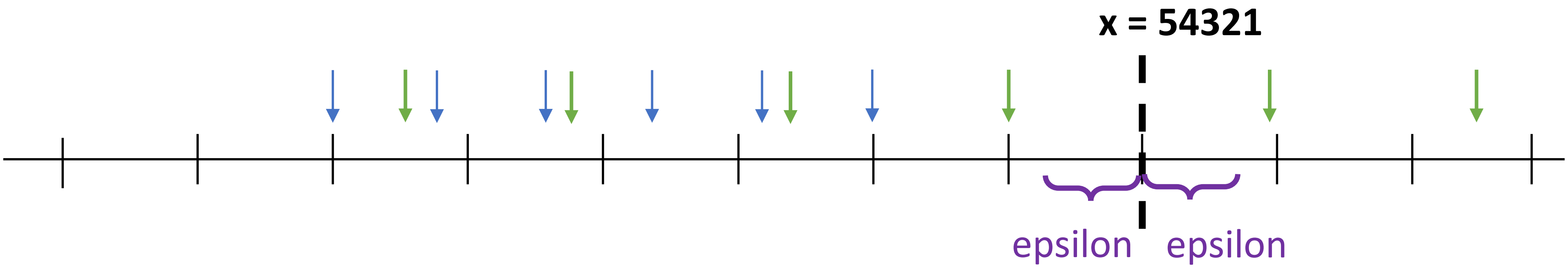
Ana Bell

LAST LECTURE

- Floating point numbers introduce challenges!
- They **can't be represented in memory exactly**
  - Operations on floats introduce tiny errors
  - Multiple operations on floats **magnify errors** :(
- Guess-and-check enumerates ints one at a time as a solution to a problem
- **Approximation methods** enumerate using a float increment. Checking a solution is not possible. Checking whether a solution yields a **value within epsilon** is possible!

RECAP: SQUARE ROOT FINDING: STOPPING CONDITION with a BIG INCREMENT (0.01)

- Blue arrow is the `guess`
- Green arrow is `guess**2`



RECAP of APPROXIMATION METHOD TO FIND A “close enough” SQUARE ROOT

```
x = 54321
epsilon = 0.01
num_guesses = 0
guess = 0.0
increment = 0.0001
while abs(guess**2 - x) >= epsilon and guess**2 <= x:
    guess += increment
    num_guesses += 1
print('num_guesses =', num_guesses)
if abs(guess**2 - x) >= epsilon:
    print('Failed on square root of', x)
else:
    print(guess, 'is close to square root of', x)
```

Stop when you are within epsilon, i.e. close enough

Stop when you go past the last reasonable guess

Exited the loop because further guesses are unreasonable

Exited the loop because guess**2 got close enough to x

BISECTION SEARCH

CHANCE to WIN BIG BUCKS

- Suppose I attach a hundred dollar bill to a particular page in the text book, 448 pages long
- If you can guess page in 8 or fewer guesses, you get big bucks
- If you fail, you get an F
- Would you want to play?

Your chances are about 1 in 56

- Now suppose on each guess I told you whether you were correct, or too low or too high
- Would you want to play in this case?

Your chances are about 1 in 3

BISECTION SEARCH

- Apply it to **problems with an inherent order** to the range of possible answers
- Suppose we know answer lies within some interval
  - Guess **midpoint** of interval
  - If not the answer, check if **answer is greater than or less** than midpoint
  - **Change** interval
  - Repeat
- Process **cuts set of things to check in half** at each stage
  - Exhaustive search reduces them from N to N-1 on each step
  - Bisection search reduces them from N to N/2

LOG GROWTH is BETTER

- Process cuts set of things to check in half at each stage
  - Characteristic of a logarithmic growth
- **Algorithm comparison: guess-and-check vs. bisection search**
  - Checking answer on-by-one iteratively is linear in the number of possible guesses
  - Checking answer by guessing the halfway point is logarithmic on the number of possible guesses
  - Log algorithm is much **more efficient**

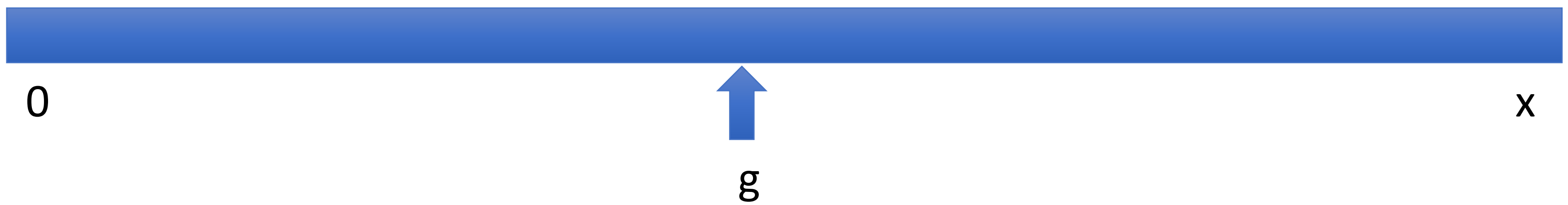
We will see discussion of relative costs of different algorithms in a few weeks

AN ANALOGY

- Suppose we forced you to sit in **alphabetical order** in class, from front left corner to back right corner
- To find a particular student, I could ask the **person in the middle** of the hall their name
- **Based on the response**, I can either dismiss the back half or the front half of the entire hall
- And I **repeat that process** until I find the person I am seeking

BISECTION SEARCH for SQUARE ROOT

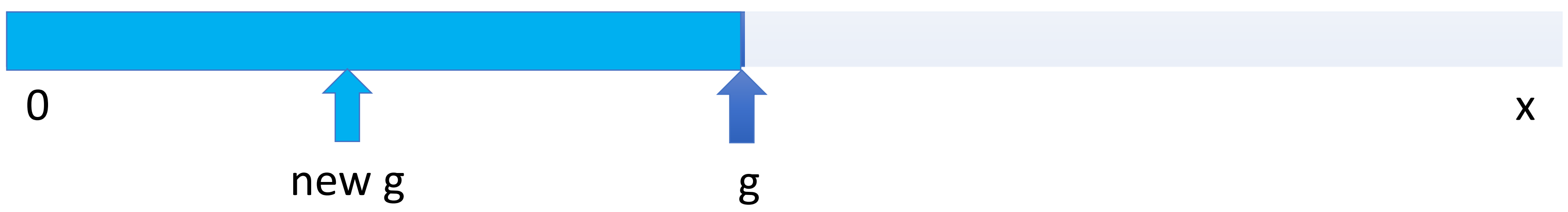
- Suppose we know that the **answer lies between 0 and x**
- Rather than exhaustively trying things starting at 0, suppose instead we **pick a number in the middle** of this range



- If we are lucky, this answer is close enough

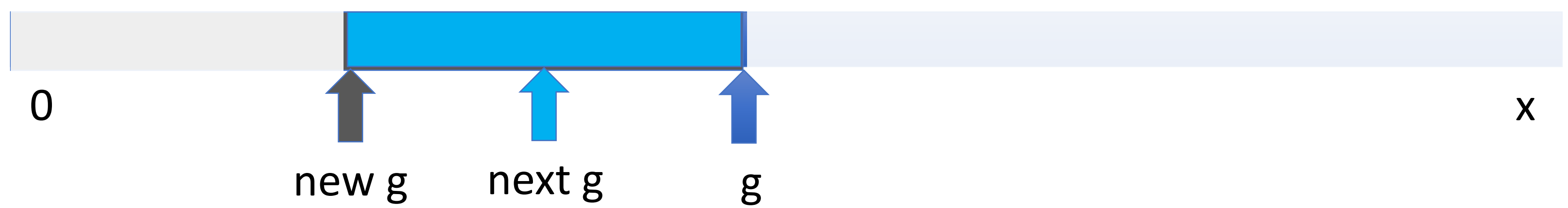
BISECTION SEARCH for SQUARE ROOT

- If not close enough, **is guess too big or too small**?
- If g**2 > x, then know g is too big; so now search



BISECTION SEARCH for SQUARE ROOT

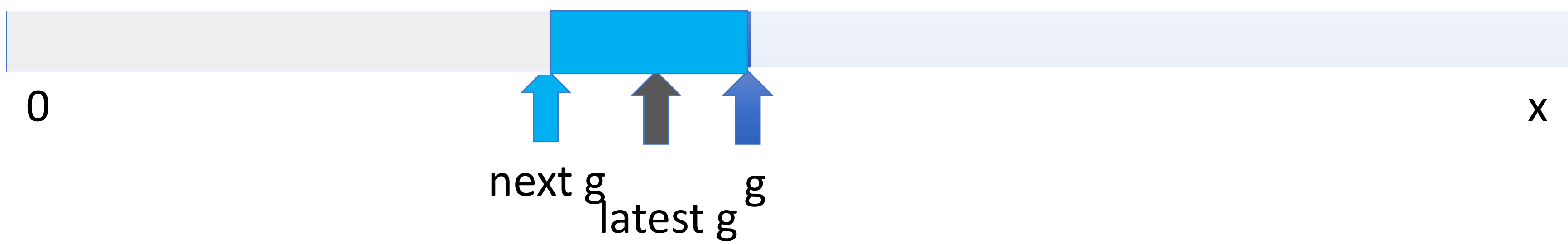
- And if, for example, this new g is such that g**2 < x, then know too small; so now search



- At each stage, **reduce range of values to search by half**

BISECTION SEARCH for SQUARE ROOT

- And if, for example, this next g is such that g**2 < x, then know too small; so now search



- At each stage, **reduce range of values to search by half**

BIG IDEA

Bisection search takes advantage of properties of the problem.

1) The search space has an order

2) We can tell whether the guess was too low or too high

YOU TRY IT!

- You are guessing a 4 digit pin code. The only feedback the phone tells you is whether your guess is correct or not. Can you use bisection search to quickly and correctly guess the code?

YOU TRY IT!

- You are playing an EXTREME guessing game to guess a number EXACTLY. A friend has a decimal number between 0 and 10 (to any precision) in mind. The feedback on your guess is whether it is correct, too high, or too low. Can you use bisection search to quickly and correctly guess the number?

SLOW SQUARE ROOT USING APPROXIMATION METHODS

```
x = 54321
epsilon = 0.01
num_guesses = 0
guess = 0.0
increment = 0.00001
while abs(guess**2 - x) >= epsilon and guess**2 <= x:
    guess += increment
    num_guesses += 1
print('num_guesses =', num_guesses)
if abs(guess**2 - x) >= epsilon:
    print('Failed on square root of', x)
else:
    print(guess, 'is close to square root of', x)
```

FAST SQUARE ROOT

```
x = 54321
epsilon = 0.01
num_guesses = 0
```

Initialize some stuff

```
while abs(guess**2 - x) >= epsilon:
```

What is repeated each time?

```
    num_guesses += 1
print('num_guesses =', num_guesses)
print(guess, 'is close to square root of', x)
```

FAST SQUARE ROOT

```
x = 54321
epsilon = 0.01
num_guesses = 0
low = 0
high = x
guess = (high + low)/2.0
while abs(guess**2 - x) >= epsilon:



    num_guesses += 1
print('num_guesses =', num_guesses)
print(guess, 'is close to square root of', x)
```

Initialize endpoints and guess

What is repeated each time?

FAST SQUARE ROOT

```
x = 54321
epsilon = 0.01
num_guesses = 0
low = 0
high = x
guess = (high + low)/2.0
while abs(guess**2 - x) >= epsilon:
    if guess**2 < x :

    else:


    num_guesses += 1
print('num_guesses =', num_guesses)
print(guess, 'is close to square root of', x)
```

Check if guess squared is too low or too high compared to x

FAST SQUARE ROOT

```
x = 54321
epsilon = 0.01
num_guesses = 0
low = 0
high = x
guess = (high + low)/2.0
while abs(guess**2 - x) >= epsilon:
    if guess**2 < x :
        low = guess
    else:


    num_guesses += 1
print('num_guesses =', num_guesses)
print(guess, 'is close to square root of', x)
```

If guess was too low, reset the low endpoint to the guess

FAST SQUARE ROOT

```
x = 54321
epsilon = 0.01
num_guesses = 0
low = 0
high = x
guess = (high + low)/2.0
while abs(guess**2 - x) >= epsilon:
    if guess**2 < x :
        low = guess
    else:
        high = guess

    num_guesses += 1
print('num_guesses =', num_guesses)
print(guess, 'is close to square root of', x)
```

If guess was too high, reset the high endpoint to the guess

FAST SQUARE ROOT
Python Tutor LINK

```
x = 54321
epsilon = 0.01
num_guesses = 0
low = 0
high = x
guess = (high + low)/2.0
while abs(guess**2 - x) >= epsilon:
    if guess**2 < x :
        low = guess
    else:
        high = guess
    guess = (high + low)/2.0
    num_guesses += 1
print('num_guesses =', num_guesses)
print(guess, 'is close to square root of', x)
```

Make a new guess using the new endpoints

LOG GROWTH is BETTER

- Brute force search for root of 54321 took over **23M guesses**
- With bisection search, reduced to **30 guesses**!
- We'll spend more time on this later, but we say the brute force method is **linear in size of problem**, because number to steps grows linearly as we increase problem size
- Bisection search is **logarithmic in size of problem**, because number of steps grows logarithmically with problem size
  - search space
    - first guess: $N/2$
    - second guess: $N/4$
    - k^{th} guess: $N/2^k$
  - guess converges on the order of $\log_2 N$ steps

WHY?

- $N/2^k = 1$ Since at this point we have one guess left to check **this tells us n in terms of k**
- $N = 2^k$ Solve this for k
- $k = \log(N)$ Tells us **k in terms of N**

It takes us k steps to guess using bisection search

==

It takes us log(N) steps to guess using bisection search

DOES IT ALWAYS WORK?

- Try running code for x such that 0 < x < 1
- If x < 1, we are **searching from 0 to x**
- But know **square root is greater than x and less than 1**
- Modify the code to choose the search space depending on value of x

You Try It: BISECTION SEARCH – SQUARE ROOT with 0 < x < 1

Choose the appropriate endpoints

```
x = 0.5
epsilon = 0.01

guess = (high + low)/2

while abs(guess**2 - x) >= epsilon:
    if guess**2 < x:
        low = guess
    else:
        high = guess
    guess = (high + low)/2.0

print(f'{str(guess)} is close to square root of {str(x)}')
```

BISECTION SEARCH – SQUARE ROOT for ALL x VALUES

```
x = 0.5
epsilon = 0.01

if x >= 1:
    low = 1.0
    high = x
else:
    low = x
    high = 1.0
guess = (high + low)/2

while abs(guess**2 - x) >= epsilon:
    if guess**2 < x:
        low = guess
    else:
        high = guess
    guess = (high + low)/2.0

print(f'{str(guess)} is close to square root of {str(x)}')
```

SOME OBSERVATIONS

- Bisection search **radically reduces computation time** – being smart about generating guesses is important
- Search space gets **smaller quickly at the beginning** and then more slowly (in absolute terms, but not as a fraction of search space) later
- Works on **problems with "ordering" property**

YOU TRY IT!

- Write code to do bisection search to find the cube root of positive cubes within some epsilon. Start with:

```
cube = 27
epsilon = 0.01
low = 0
high = cube
```

NEWTON-RAPHSON

- General **approximation algorithm to find roots of a polynomial** in one variable

$$p(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0$$

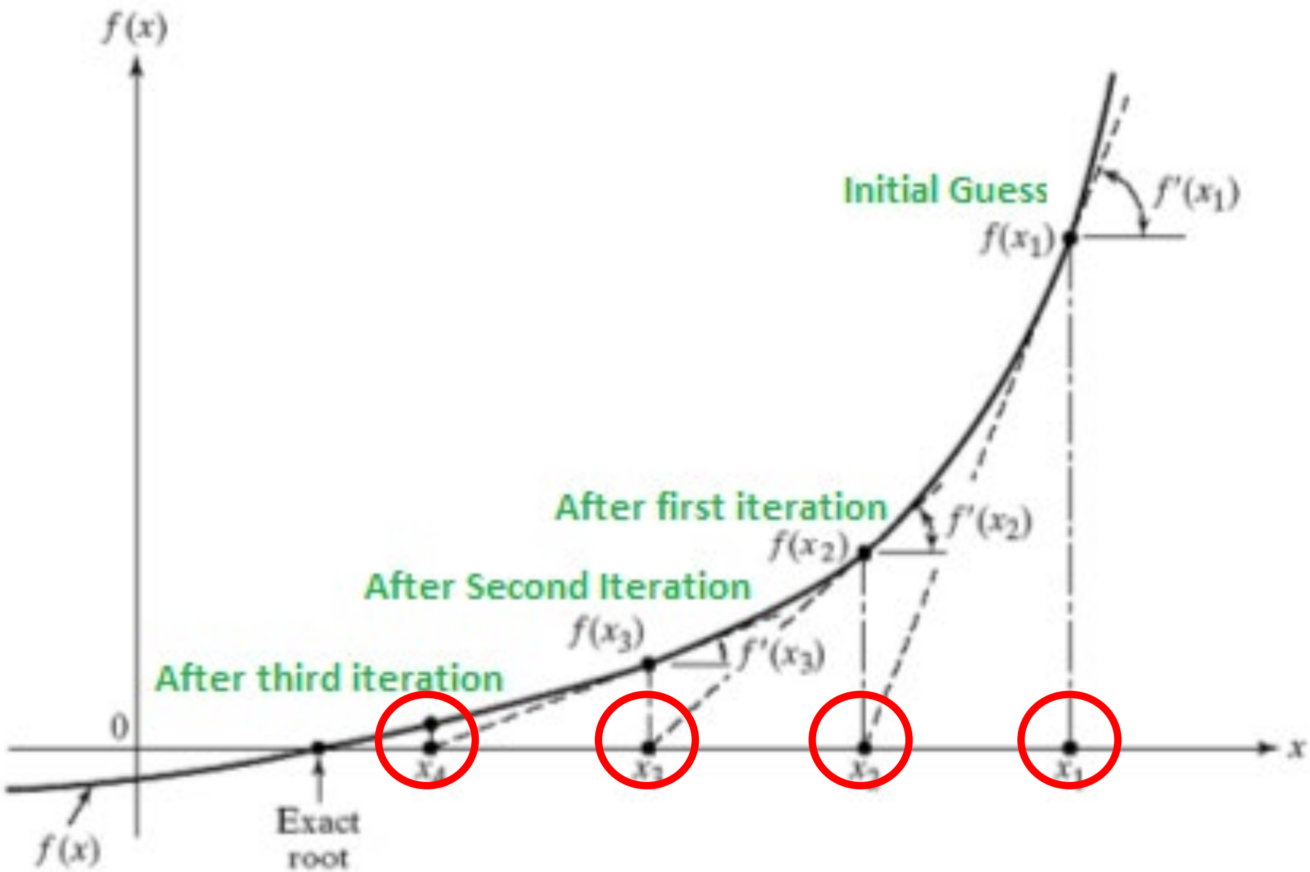
- Newton and Raphson showed that if g is an approximation to the root, then

$$g - p(g)/p'(g)$$

  is a better approximation; where p' is derivative of p

- Try to use this idea for **finding the square root of x**
  - Want to find r such that p(r) = 0
  - For example, to find the square root of 24, find the root of $p(x) = x^2 - 24$

INTUITION - LINK



NEWTON-RAPHSON ROOT FINDER

- Simple case for a polynomial: $x^2 - k$
- First derivative: $2x$
- Newton-Raphson says given a guess g for root of k, a better guess is:

$$g - (g^2 - k)/2g$$

- This eventually **finds an approximation to the square root of k!**

NEWTON-RAPHSON ROOT FINDER

- Another **way of generating guesses** which we can check; very efficient

$f(x) = x^2 - 24$

```
epsilon = 0.01
k = 24.0
guess = k/2.0
num_guesses = 0

while abs(guess*guess - k) >= epsilon:
    num_guesses += 1
    guess = guess - (((guess**2) - k)/(2*guess))
print('num_guesses = ' + str(num_guesses))
print('Square root of ' + str(k) + ' is about ' + str(guess))
```

f(guess)

f'(guess)

ITERATIVE ALGORITHMS

- Guess and check methods build on **reusing same code**
  - Use a looping construct
  - Generate guesses (important difference in algorithms)
  - Check and continue
- **Generating guesses**
  - Exhaustive enumeration
  - Approximation algorithm
  - Bisection search
  - Newton-Raphson (for root finding)

SUMMARY

- For many problems, **cannot find exact answer**
- Need to seek a **“good enough” answer** using approximations
- When testing floating point numbers
  - It’s important to understand how the computer represents these in binary
  - Understand why we use “close enough” and not “==“
- Bisection search works is FAST but for problems with:
  - Two **endpoints**
  - An **ordering** to the values
  - **Feedback** on guesses (too low, too high, correct, etc.)
- Newton-Raphson is a smart way to find roots of a polynomial

DECOMPOSITION and ABSTRACTION

LEARNING to CREATE CODE

- So far have covered **basic language mechanisms** – primitives, complex expressions, branching, iteration
- In principle, **you know all you need to know** to accomplish anything that can be done by computation
- But in fact, we've taught you **nothing** about two of the most important concepts in programming…

DECOMPOSITION and ABSTRACTION

- **Decomposition**
- How to divide a program into **self-contained parts** that can be combined to solve the current problem

DECOMPOSITION and ABSTRACTION

- **Abstraction**
- How to ignore **unnecessary detail**

DECOMPOSITION and ABSTRACTION

- Decomposition:
  - Ideally **parts can be reused** by other programs
  - Self-contained means parts should **complete computation using only inputs provided** to them and "basic" operations

```
a = 3.14*2.2*2.2
```

```
pi = 3.14
r = 2.2
area = pi*r**2
```

- Abstraction:
  - Used to separate **what** something does, from **how** it actually does it
  - Creating parts and abstracting away details allows us to write complex code while **suppressing details**, so that we are not overwhelmed by that complexity

```
# calculate the area of a circle
```

BIG IDEA

Make code easy to

create
modify
maintain
understand

DECOMPOSITION, ABSTRACTION, FUNCTIONS

(download slides and .py files to follow along)

6.100L Lecture 7

Ana Bell

AN EXAMPLE: the SMARTPHONE

- A black box, and can be viewed in terms of
  - Its inputs
  - Its outputs
  - How outputs are related to inputs, without any knowledge of its internal workings
  - Implementation is “opaque” (or black)

AN EXAMPLE: the SMARTPHONE ABSTRACTION

- User **doesn't know the details** of how it works
  - We **don't need to know how something works** in order to know how to use it
- User **does know the interface**
  - Device converts a sequence of screen touches and sounds into expected useful functionality
- Know **relationship** between input and output

ABSTRACTION ENABLES DECOMPOSITION

- 100's of distinct parts
- Designed and made by different companies
  - Do not communicate with each other, other than specifications for components
  - May use same subparts as others
- Each component maker has to know **how its component interfaces** to other components
- Each component maker can **solve sub-problems independent of other parts**, so long as they provide specified inputs
- True for hardware and for software

BIG IDEA

Apply
abstraction (black box) and
decomposition (split into self-contained parts)
to programming!

SUPPRESS DETAILS with ABSTRACTION

- In programming, want to think of piece of code as **black box**
  - Hide tedious coding details from the user
  - Reuse black box at different parts in the code (no copy/pasting!)
- **Coder creates details**, and designs interface
- **User** does **not need or want** to see details

SUPPRESS DETAILS with ABSTRACTION

- Coder achieves abstraction with a **function (or procedure)**
- You've already been using functions!
- **A function** lets us capture code within a black box
  - Once we create function, it will produce an output from inputs, while hiding details of how it does the computation

```
max(1,4)
abs(-3)
len("mom's spaghetti")
```

SUPPRESS DETAILS with ABSTRACTION

- A function has **specifications**, captured using **docstrings**
- Think of a **docstring** as "contract" between coder and user:
  - If user provides **input** that satisfies stated conditions, function will produce **output** according to specs, including indicated **side effects**
  - Not typically enforced in Python (we'll see assertions later), but user relies on coder's work satisfying the contract

```
abs(-3)
```

```
abs(
```

```
abs(x, /)

Return the absolute value of the argument.
```

CREATE STRUCTURE with DECOMPOSITION

- Given the idea of black box abstraction, use it to **divide code into modules** that are:
  - **Self-contained**
  - Intended to be **reusable**
- Modules are used to:
  - **Break up** code into logical pieces
  - Keep code **organized**
  - Keep code **coherent** (readable and understandable)
- In this lecture, achieve decomposition with **functions**
- In a few lectures, achieve decomposition with **classes**
- Decomposition relies on abstraction to enable construction of complex modules from simpler ones

FUNCTIONS

- Reusable pieces of code, called **functions** or **procedures**
- Capture steps of a computation so that we can use with any input
- A function is just some **code written in a special, reusable way**

FUNCTIONS

- **Defining a function** tells Python some code now exists in memory
- Functions are only useful when they are **run** ("**called**" or "**invoked**")
- You write a function once but can run it many times!
- Compare to code in a file
  - It doesn't run when you load the file
  - It runs when you hit the run button

FUNCTION CHARACTERISTICS

- Has a **name**
  - (think: variable bound to a function object)
- Has (formal) **parameters** (0 or more)
  - The inputs
- Has a **docstring** (optional but recommended)
  - A comment delineated by `"""` (triple quotes) that provides a **specification** for the function – contract relating output to input
- Has a **body**, a set of instructions to execute when function is called
- **Returns** something
  - Keyword `return`

HOW to WRITE a FUNCTION

keyword name parameters or arguments specification, docstring

```
def is_even( i ):
    """
    Input: i, a positive int
    Returns True if i is even, otherwise False
    """
    if i%2 == 0:
        return True
    else:
        return False
```

indentation defines extent of function body

body

return keyword tells function to give back a value

HOW TO THINK ABOUT WRITING A FUNCTION

- **What** is the problem?
  - Given an int, call it `i`, want to know if it is even
  - Use this to write the function name and specs

```
def is_even( i ):
    """
    Input: i, a positive int
    Returns True if i is even, otherwise False
    """
```

HOW TO THINK ABOUT WRITING A FUNCTION

- How to **solve** the problem?
  - Can check that remainder when divided by 2 is 0
  - Think about what value you need to give back

```
def is_even( i ):
    """
    Input: i, a positive int
    Returns True if i is even, otherwise False
    """
    if i%2 == 0:
        return True
    else:
        return False
```

HOW TO THINK ABOUT WRITING A FUNCTION

- Can you make the code **cleaner**?
  - i%2 is a Boolean that evaluates to True/False already

```
def is_even( i ):
    """
    Input: i, a positive int
    Returns True if i is even, otherwise False
    """
    return i%2 == 0
```

BIG IDEA

At this point, all we’ve done is make a function object

HOW TO CALL (INVOKE) A FUNCTION

Name of the function

Values for parameters of the function

```
is_even(3)
is_even(8)
```

- That's all!

HOW TO CALL (INVOKE) A FUNCTION

```
is_even(3)
```

Replaced by the return!

```
is_even(8)
```

- That's all!

ALL TOGETHER IN A FILE

- This code might be in one file

Function definition

```
def is_even( i ):
    return i%2 == 0
```

Function call

```
is_even(3)
```

WHAT HAPPENS when you CALL a FUNCTION?

- Python replaces:
  formal parameters in function def with **values from function call**
  i replaced with **3**

i mapped to 3

```
def is_even( i ):
    return i%2 == 0
```

```
is_even(3)
```

WHAT HAPPENS when you CALL a FUNCTION?

- Python replaces:
  formal parameters in function def with **values from function call**
  i replaced with **3**
- Python **executes expressions in the body** of the function
  - `return 3%2 == 0`

```
def is_even( i ):
    return i%2 == 0
```

keyword

expression to evaluate and return to invoker

`3%2 == 0` is `False`

```
is_even(3)
```

Replaced by False

WHAT HAPPENS when you CALL a FUNCTION?

- Python replaces:
  formal parameters in function def with **values from function call**
  i replaced with **3**

```
def is_even( i ):
    return i%2 == 0
```

```
is_even(3)
print(is_even(3))
```

Replaced by False

BIG IDEA

A function’s code only runs when you call (aka invoke) the function

YOU TRY IT!

- Write code that satisfies the following specs

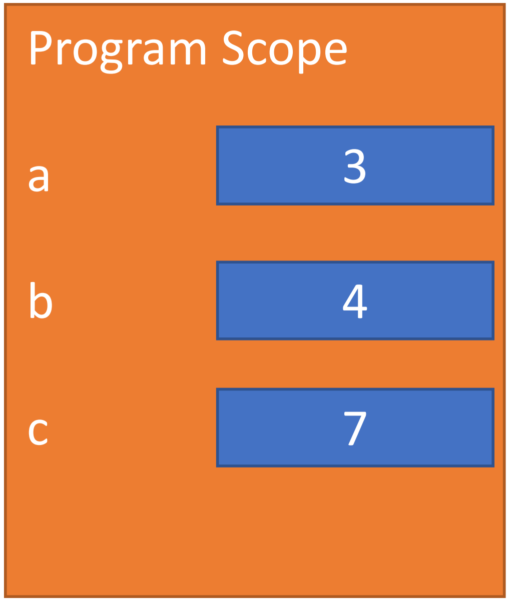
```
def div_by(n, d):
    """ n and d are ints > 0
        Returns True if d divides n evenly and False otherwise """
```

Test your code with:

- n = 10 and d = 3
- n = 195 and d = 13

ZOOMING OUT (no functions)

```
a = 3
b = 4
c = a+b
```



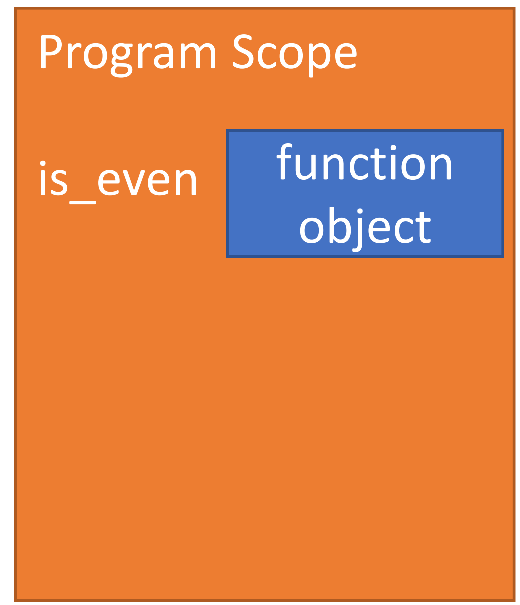
ZOOMING OUT

This is my “black box”

```python
def is_even( i ):
    print("inside is_even")
    return i%2 == 0
```

```python
a = is_even(3)
b = is_even(10)
c = is_even(123456)
```

This is me telling my black box to do something



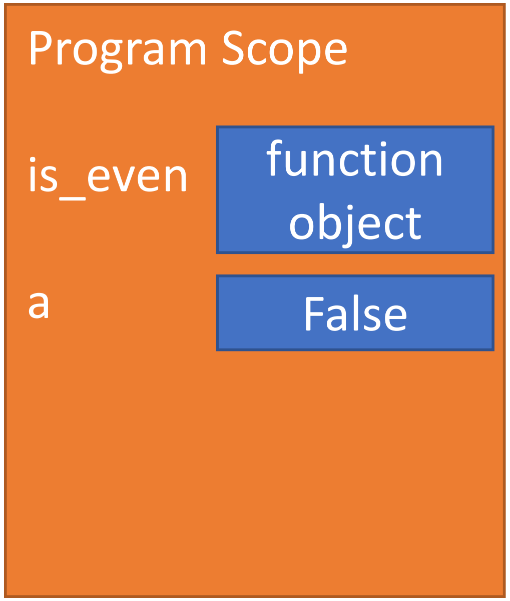
ZOOMING OUT

This is my “black box”

```
def is_even( i ):
    print("inside is_even")
    return i%2 == 0
```

```
a = is_even(3)
b = is_even(10)
c = is_even(123456)
```

One function call



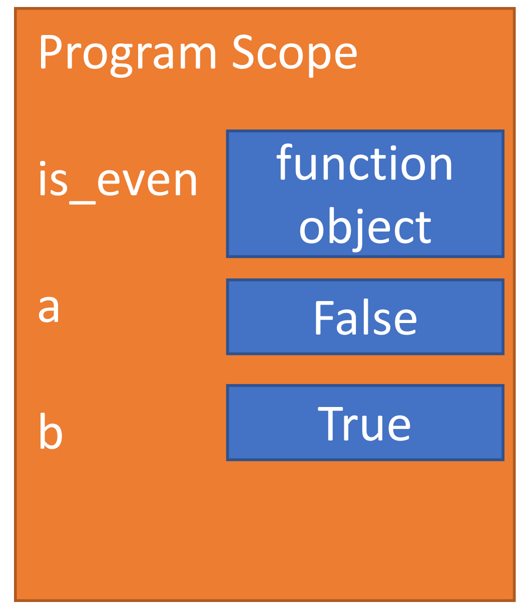
ZOOMING OUT

This is my “black box”

```
def is_even( i ):
    print("inside is_even")
    return i%2 == 0
```

```
a = is_even(3)
b = is_even(10)
c = is_even(123456)
```

One function call



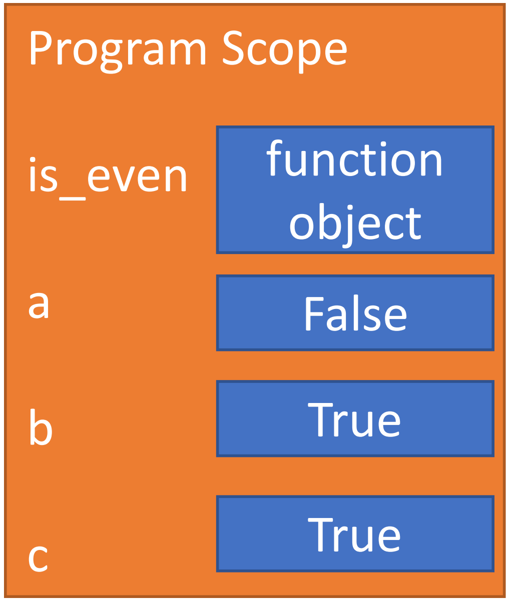
ZOOMING OUT

This is my “black box”

```
def is_even( i ):
    print("inside is_even")
    return i%2 == 0
```

```
a = is_even(3)
b = is_even(10)
c = is_even(123456)
```

One function call



INSERTING FUNCTIONS IN CODE

- Remember how expressions are replaced with the value?
- The **function call** is **replaced** with the **return value**!

```
print("Numbers between 1 and 10: even or odd")

for i in range(1,10):
    if is_even(i):
        print(i, "even")
    else:
        print(i, "odd")
```

ANOTHER EXAMPLE

- Suppose we want to add all the odd integers between (and including) `a` and `b`

- What is the **input**?
  - Values for a and b
- What is the **output**?
  - The sum_of_odds

```
def sum_odd(a, b):
    # your code here
    return sum_of_odds
```

BIG IDEA

Don't write code right away!

PAPER FIRST

- Suppose we want to add all the odd integers between (and including) `a` and `b`

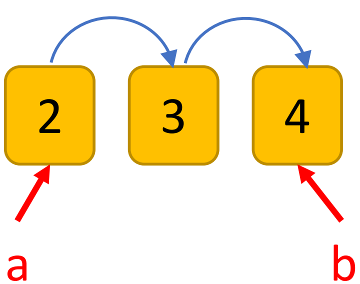
- Start with a **simple example on paper**
- Systematically solve the example

```
def sum_odd(a, b):
    # your code here
    return sum_of_odds
```

SIMPLE TEST CASE

- Suppose we want to add all the odd integers between (and including) `a` and `b`

- Start with a simple example on paper
- a = 2 and b = 4
  - sum_of_odds should be 3

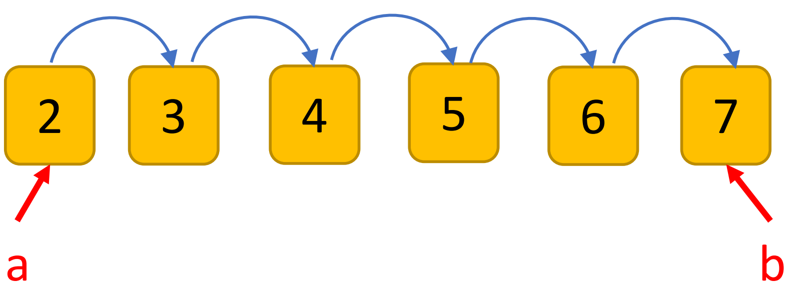



```
def sum_odd(a, b):
    # your code here
    return sum_of_odds
```

MORE COMPLEX TEST CASE

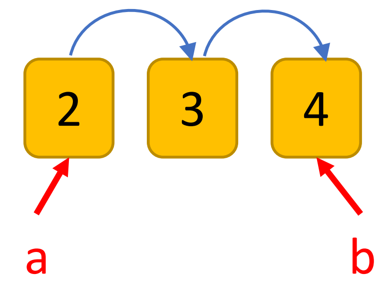
- Suppose we want to add all the odd integers between (and including) `a` and `b`

- Start with a simple example on paper
- a = 2 and b = 7
  - sum_of_odds should be 15




```
def sum_odd(a, b):
    # your code here
    return sum_of_odds
```

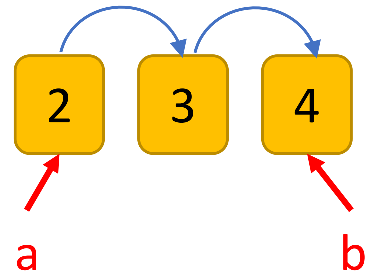
SOLVE SIMILAR PROBLEM



- Start by looking at each number between (and including) a and b
- A similar problem that is easier that you know how to do?
  - Add **ALL** numbers between (and including) a and b
  - Start with this

```
def sum_odd(a, b):
    # your code here
    return sum_of_odds
```

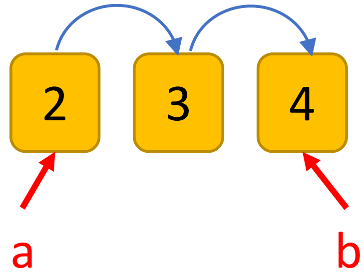
CHOOSE BIG-PICTURE STRUCTURE



- Add **ALL** numbers between (and including) a and b
  - It's a loop
- while or for?
  - Your choice

```
def sum_odd(a, b):
    # your code here
    return sum_of_odds
```

WRITE the LOOP
(for adding all numbers)



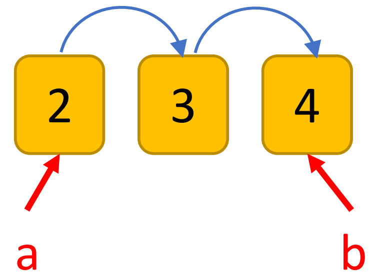
`for` **LOOP**

```
def sum_odd(a, b):
    for i in range(a, b):
        # do something
    return sum_of_odds
```

`while` **LOOP**

```
def sum_odd(a, b):
    i = a
    while i <= b:
        # do something
        i += 1
    return sum_of_odds
```

DO the SUMMING (for adding all numbers)



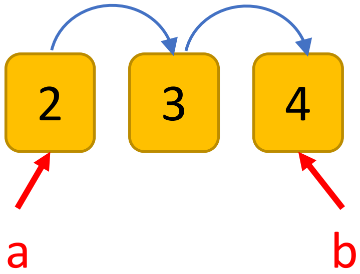
`for` LOOP

```
def sum_odd(a, b):
    for i in range(a, b):
        sum_of_odds += i
    return sum_of_odds
```

`while` LOOP

```
def sum_odd(a, b):
    i = a
    while i <= b:
        sum_of_odds += i
        i += 1
    return sum_of_odds
```

INITIALIZE the SUM (for adding all numbers)



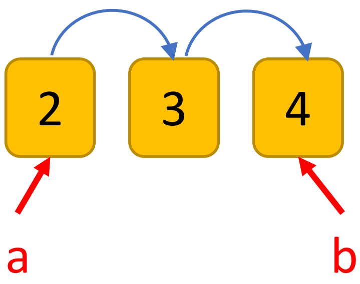
`for` LOOP

```
def sum_odd(a, b):
    sum_of_odds = 0
    for i in range(a, b):
        sum_of_odds += i
    return sum_of_odds
```

`while` LOOP

```
def sum_odd(a, b):
    sum_of_odds = 0
    i = a
    while i <= b:
        sum_of_odds += i
        i += 1
    return sum_of_odds
```

TEST!
(for adding all numbers)



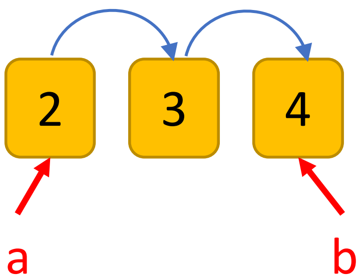
`for` LOOP

```
def sum_odd(a, b):
    sum_of_odds = 0
    for i in range(a, b):
        sum_of_odds += i
    return sum_of_odds



print(sum_odd(2,4))
```

`while` LOOP

```
def sum_odd(a, b):
    sum_of_odds = 0
    i = a
    while i <= b:
        sum_of_odds += i
        i += 1
    return sum_of_odds


print(sum_odd(2,4))
```

WEIRD RESULTS…
(for adding all numbers)



`for` LOOP

```
def sum_odd(a, b):
    sum_of_odds = 0
    for i in range(a, b):
        sum_of_odds += i
    return sum_of_odds


print(sum_odd(2,4))
```

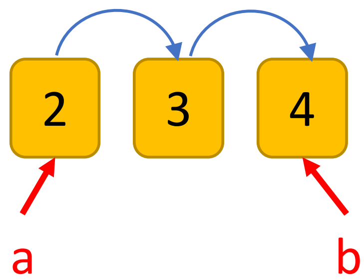
5

`while` LOOP

```
def sum_odd(a, b):
    sum_of_odds = 0
    i = a
    while i <= b:
        sum_of_odds += i
        i += 1
    return sum_of_odds


print(sum_odd(2,4))
```

9

DEBUG! aka ADD PRINT STATEMENTS (for adding all numbers)



`for` LOOP

```
def sum_odd(a, b):
    sum_of_odds = 0
    for i in range(a, b):
        sum_of_odds += i
        print(i, sum_of_odds)
    return sum_of_odds
```

```
print(sum_odd(2,4))
```

```
2 2
3 5
```

5

`while` LOOP

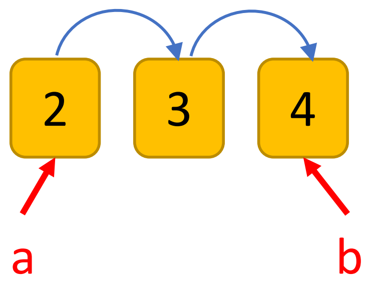
```
def sum_odd(a, b):
    sum_of_odds = 0
    i = a
    while i <= b:
        sum_of_odds += i
        print(i, sum_of_odds)
        i += 1
    return sum_of_odds
```

```
print(sum_odd(2,4))
```

```
2 2
3 5
4 9
```

9

FIX for LOOP END INDEX
(for adding all numbers)



`for` LOOP

```
def sum_odd(a, b):
    sum_of_odds = 0
    for i in range(a, b+1):
        sum_of_odds += i
        print(i, sum_of_odds)
    return sum_of_odds


print(sum_odd(2,4))
```

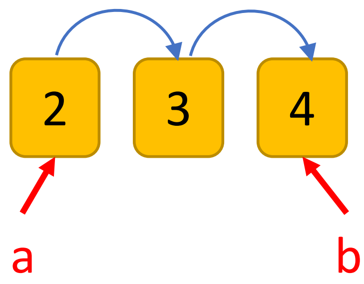
9

`while` LOOP

```
def sum_odd(a, b):
    sum_of_odds = 0
    i = a
    while i <= b:
        sum_of_odds += i
        print(i, sum_of_odds)
        i += 1
    return sum_of_odds

print(sum_odd(2,4))
```

9

ADD IN THE ODD PART!



`for` LOOP

```
def sum_odd(a, b):
    sum_of_odds = 0
    for i in range(a, b+1):
        if i%2 == 1:
            sum_of_odds += i
            print(i, sum_of_odds)
    return sum_of_odds

print(sum_odd(2,4))
```

3

`while` LOOP

```
def sum_odd(a, b):
    sum_of_odds = 0
    i = a
    while i <= b:
        if i%2 == 1:
            sum_of_odds += i
            print(i, sum_of_odds)
        i += 1
    return sum_of_odds
print(sum_odd(2,4))
```

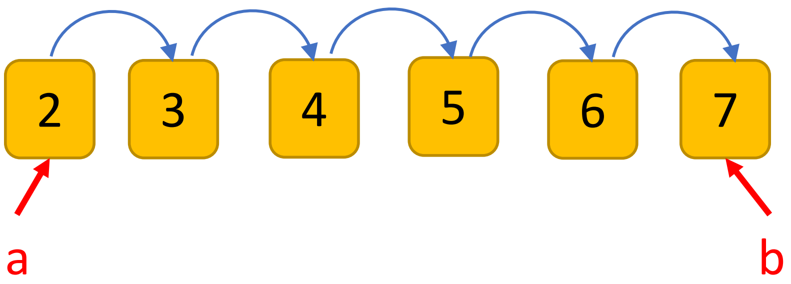
3

BIG IDEA

Solve a simpler problem first.

Add functionality to the code later.

TRY IT ON ANOTHER EXAMPLE



`for` LOOP

```
def sum_odd(a, b):
    sum_of_odds = 0
    for i in range(a, b+1):
        if i%2 == 1:
            sum_of_odds += i
    return sum_of_odds


print(sum_odd(2,7))
```

15

`while` LOOP

```
def sum_odd(a, b):
    sum_of_odds = 0
    i = a
    while i <= b:
        if i%2 == 1:
            sum_of_odds += i
        i += 1
    return sum_of_odds

print(sum_odd(2,7))
```

15

PYTHON TUTOR

- Also a great debugging tool

BIG IDEA

Test code often.
Use prints to debug.

YOU TRY IT!

- Write code that satisfies the following specs

```
def is_palindrome(s):
    """ s is a string
    Returns True if s is a palindrome and False otherwise
    """
```

For example:

- If s = "222" returns True
- If s = "2222" returns True
- If s = "abc" returns False

SUMMARY

- Functions allow us to **suppress detail** from a user
- Functions **capture computation** within a black box
- A programmer writes functions with
  - 0 or more **inputs**
  - Something to **return**
- A function only **runs when it is called**
- The entire function call is **replaced with the return value**
  - Think expressions! And how you replace an entire expression with the value it evaluates to.

FUNCTIONS as OBJECTS

(download slides and .py files to follow along)

6.100L Lecture 8

Ana Bell

FUNCTION FROM LAST LECTURE

```
def is_even( i ):
    """
    Input: i, a positive int
    Returns True if i is even and False otherwise
    """
    return i%2 == 0
```

- A function always returns something

WHAT IF THERE IS NO `return` KEYWORD

```
def is_even( i ):
    """
    Input: i, a positive int
    Does not return anything
    """
    i%2 == 0
```

without a return statement

- Python returns the value **None, if no `return` given**
- Represents the absence of a value
    - If invoked in shell, nothing is printed
- No static semantic error generated

```
def is_even( i ):
    """
    Input: i, a positive int
    Does not return anything
    """
    i%2 == 0
    return None
```

None is a value of type NoneType (not a string, not a number, etc)

A line Python adds implicitly (do not add it yourself)

YOU TRY IT!

- What is printed if you run this code as a file?

```
def add(x,y):
    return x+y
def mult(x,y):
    print(x*y)


add(1,2)
print(add(2,3))
mult(3,4)
print(mult(4,5))
```

`return` vs. `print`

- return only has meaning **inside** a function
- only **one** return executed inside a function
- code inside function, but after return statement, not executed
- has a value associated with it, **given to function caller**

- print can be used **outside** functions
- can execute **many** print statements inside a function
- code inside function can be executed after a print statement
- has a value associated with it, **outputted** to the console
- print expression itself returns `None` value

YOU TRY IT!

- Fix the code that tries to write this function

```
def is_triangular(n):
    """ n is an int > 0
    Returns True if n is triangular, i.e. equals a continued
    summation of natural numbers (1+2+3+...+k), False otherwise """
    total = 0
    for i in range(n):
        total += i
        if total == n:
            print(True)
    print(False)
```

FUNCTIONS SUPPORT MODULARITY

- Here is our bisection square root method as a function

```
def bisection_root(x):
    epsilon = 0.01
    low = 0
    high = x
    ans = (high + low)/2.0
    while abs(ans**2 - x) >= epsilon:
        if ans**2 < x:
            low = ans
        else:
            high = ans
        ans = (high + low)/2.0
    # print(ans, 'is close to the root of', x)
    return ans
```

Initialize variables

guess not close enough

iterate

update low or high, depends on guess too small or too large

new value for guess

return result

FUNCTIONS SUPPORT MODULARITY

- Call it with different values

```
print(bisection_root(4))
print(bisection_root(123))
```

- Write a function that calls this one!

YOU TRY IT!

- Write a function that satisfies the following specs

```
def count_nums_with_sqrt_close_to (n, epsilon):
    """ n is an int > 2
        epsilon is a positive number < 1
    Returns how many integers have a square root within epsilon of n """
```

Use `bisection_root` we already wrote to get an approximation for the sqrt of an integer.

For example: `print(count_nums_with_sqrt_close_to(10, 0.1))`

prints 4 because all these integers have a sqrt within 0.1

- sqrt of `99` is `9.949699401855469`
- sqrt of `100` is `9.999847412109375`
- sqrt of `101` is `10.049758911132812`
- sqrt of `102` is `10.09945678710937 5`

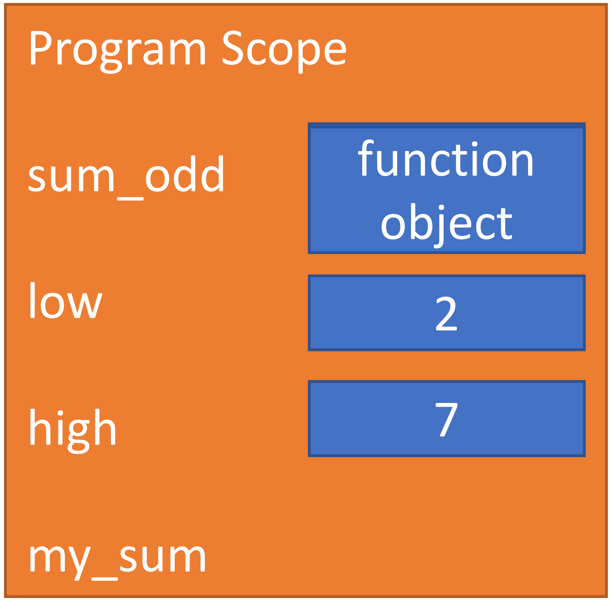
ZOOMING OUT

This is my “black box”

```python
def sum_odd(a, b):
    sum_of_odds = 0
    for i in range(a, b+1):
        if i%2 == 1:
            sum_of_odds += i
    return sum_of_odds
```

```python
low = 2
high = 7
my_sum = sum_odd(low, high)
```

One function call

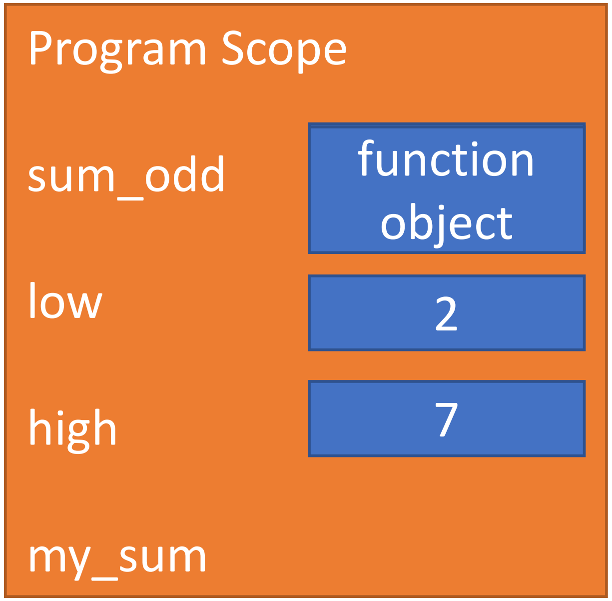


ZOOMING OUT

a = 2   b = 7

```python
def sum_odd(a, b):
    sum_of_odds = 0
    for i in range(a, b+1):
        if i%2 == 1:
            sum_of_odds += i
    return sum_of_odds
```

```python
low = 2
high = 7
my_sum = sum_odd(low, high)
```

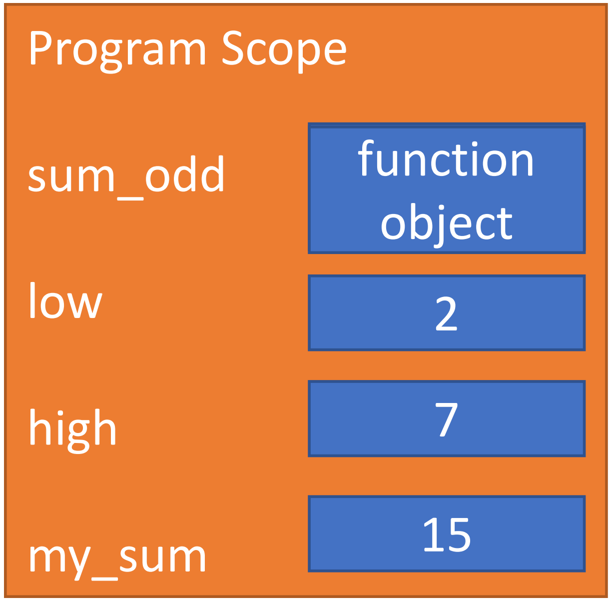


ZOOMING OUT

This is my “black box”

```
def sum_odd(a, b):
    sum_of_odds = 0
    for i in range(a, b+1):
        if i%2 == 1:
            sum_of_odds += i
    return sum_of_odds

low = 2
high = 7
my_sum = sum_odd(low, high)
```

15



FUNCTION SCOPE

UNDERSTANDING FUNCTION CALLS

- How does Python execute a function call?
- How does Python know what value is associated with a variable name?
- It **creates a new environment** with every function call!
  - Like a **mini program** that it needs to complete
  - The mini program runs with **assigning its parameters** to some inputs
  - It does the work (aka the **body** of the function)
  - It **returns** a value
  - The **environment disappears** after it returns the value

ENVIRONMENTS

- **Global** environment
  - Where user interacts with Python interpreter
  - Where the program starts out
- Invoking a function creates a **new environment** (frame/scope)

VARIABLE SCOPE

- **Formal parameters** get bound to the value of **input parameters**
- **Scope** is a mapping of names to objects
  - Defines context in which body is evaluated
  - Values of variables given by bindings of names
- Expressions in body of function evaluated wrt this new scope

```
def f( x ):
    x = x + 1
    print('in f(x): x =', x)
    return x

y = 3
z = f( y )
```

formal parameter

Function definition

actual parameter

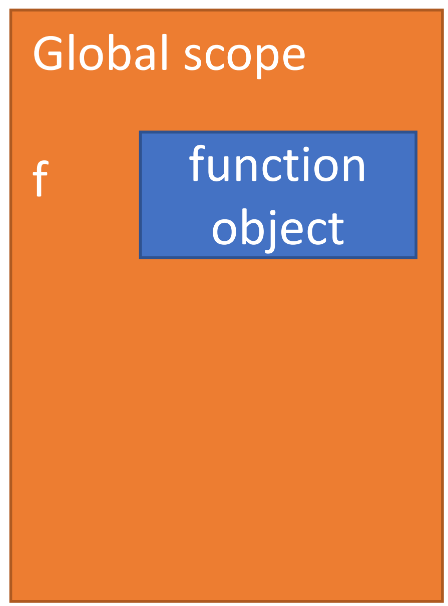
Main program code
* initializes a variable x
* makes a function call f(x)
* assigns return of function to variable z

Can be any legal value

VARIABLE SCOPE
after evaluating def

This is my “black box”

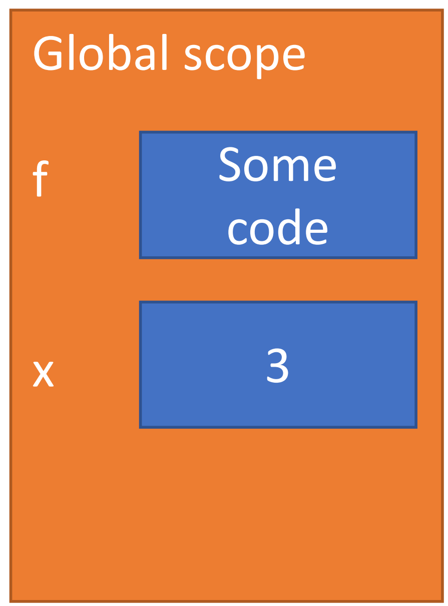
```
def f( x ):
    x = x + 1
    print('in f(x): x =', x)
    return x

x = 3
z = f( x )
```



VARIABLE SCOPE
after exec 1st assignment

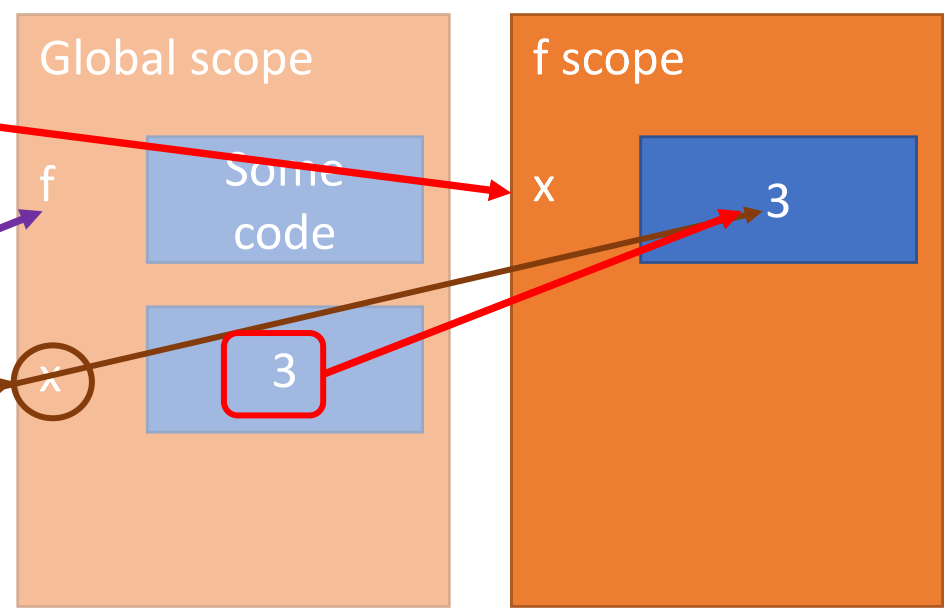
This is my “black box”

```
def f( x ):
    x = x + 1
    print('in f(x): x =', x)
    return x

x = 3
z = f( x )
```



VARIABLE SCOPE
after f invoked

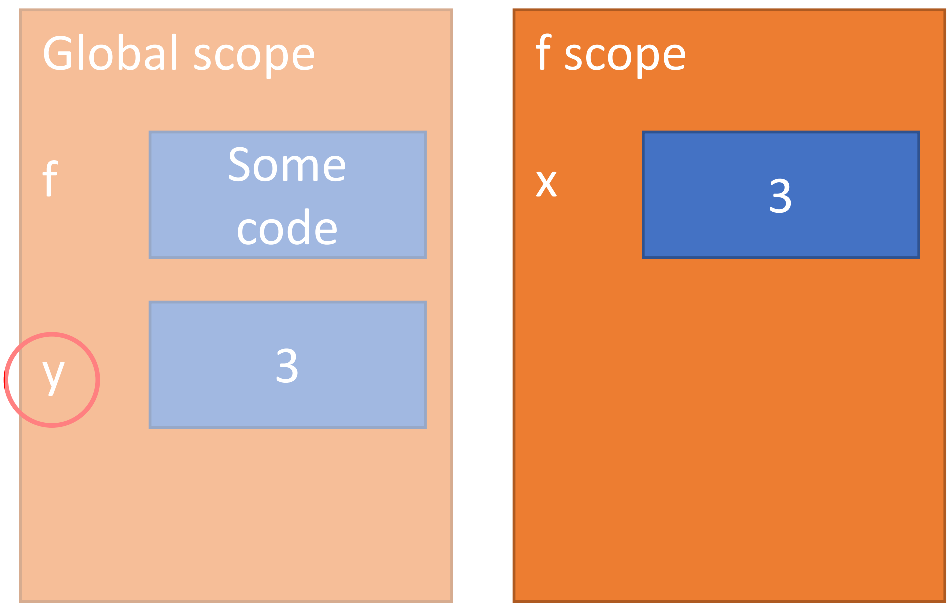
```
def f( x ):
    x = x + 1
    print('in f(x): x =', x)
    return x

x = 3
z = f( x )
```



VARIABLE SCOPE
after f invoked

Name of variable irrelevant, only value important. You can also pass in the value directly.

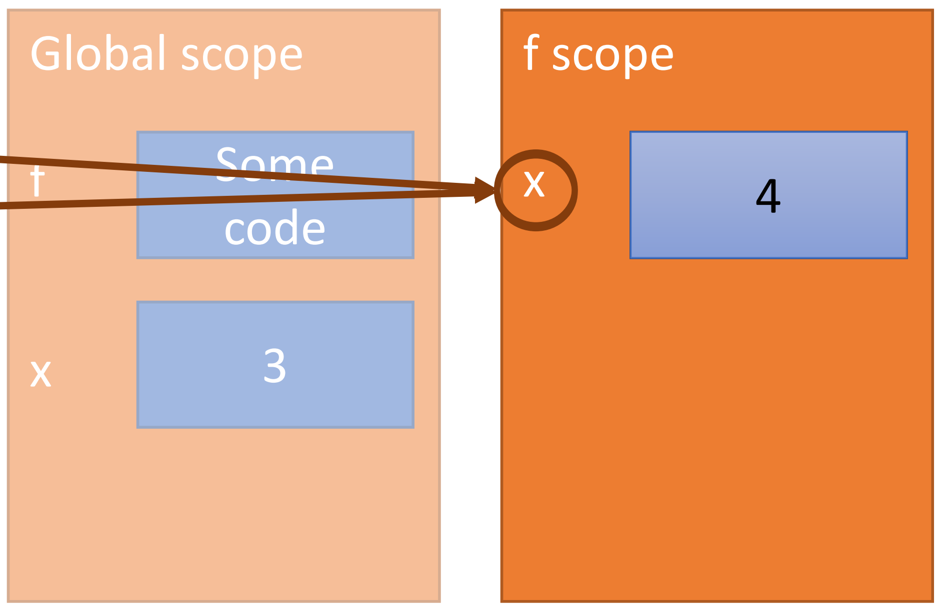
```
def f( x ):
    x = x + 1
    print('in f(x): x =', x)
    return x

y = 3
z = f( y )
```



VARIABLE SCOPE
eval body of f in f’s scope

`in f(x): x = 4` printed out

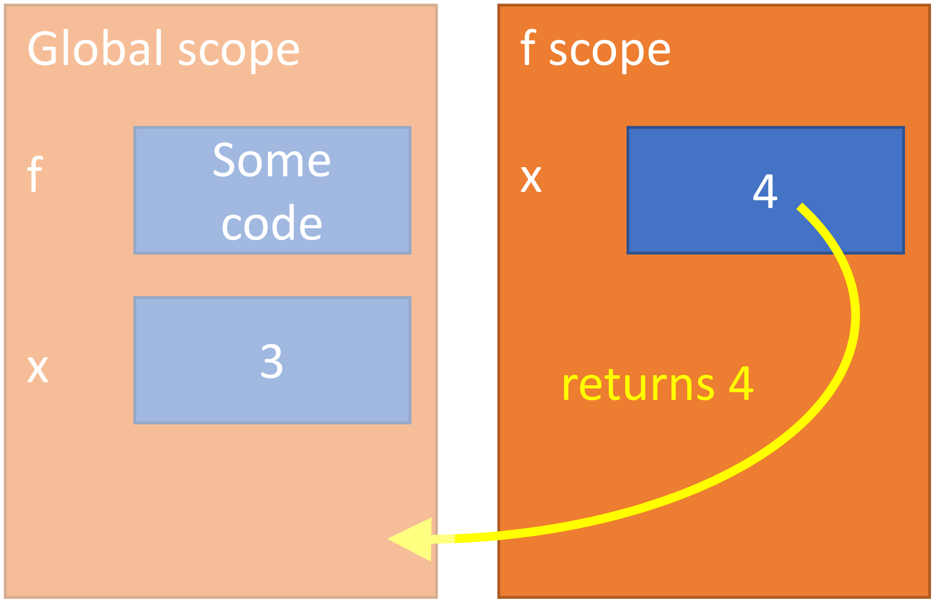
```
def f( x ):
    x = x + 1
    print('in f(x): x =', x)
    return x

x = 3
z = f( x )
```



VARIABLE SCOPE during return

```
def f( x ):
    x = x + 1
    print('in f(x): x =', x)
    return x

x = 3
z = f( x )
```

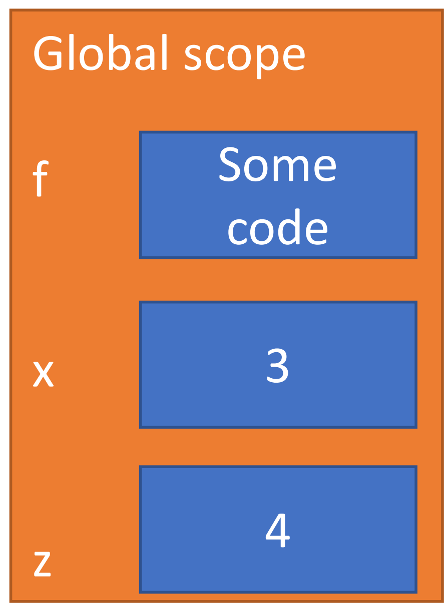
Function call replaced with return value



VARIABLE SCOPE
after exec 2nd assignment

If I now ask for value of x in
Python interpreter, it will print 3

```
def f( x ):
    x = x + 1
    print('in f(x): x =', x)
    return x

x = 3
z = f( x )
```



BIG IDEA

You need to know what expression you are executing to know the scope you are in.

ANOTHER SCOPE EXAMPLE

- Inside a function, **can access** a variable defined outside
- Inside a function, **cannot modify** a variable defined outside (can by using **global variables**, but frowned upon)
- Use the Python Tutor to step through these!

```
def f(y):
    x = 1
    x += 1
    print(x)

x = 5
f(x)
print(x)
```

x is re-defined in scope of f

different x objects

2
5

```
def g(y):
    print(x)
    print(x + 1)

x = 5
g(x)
print(x)
```

x from outside g

x inside g is picked up from scope that called function g

5
6
5

```
def h(y):
    x += 1

x = 5
h(x)
print(x)
```

UnboundLocalError: local variable 'x' referenced before assignment

Error

FUNCTIONS as ARGUMENTS

HIGHER ORDER PROCEDURES

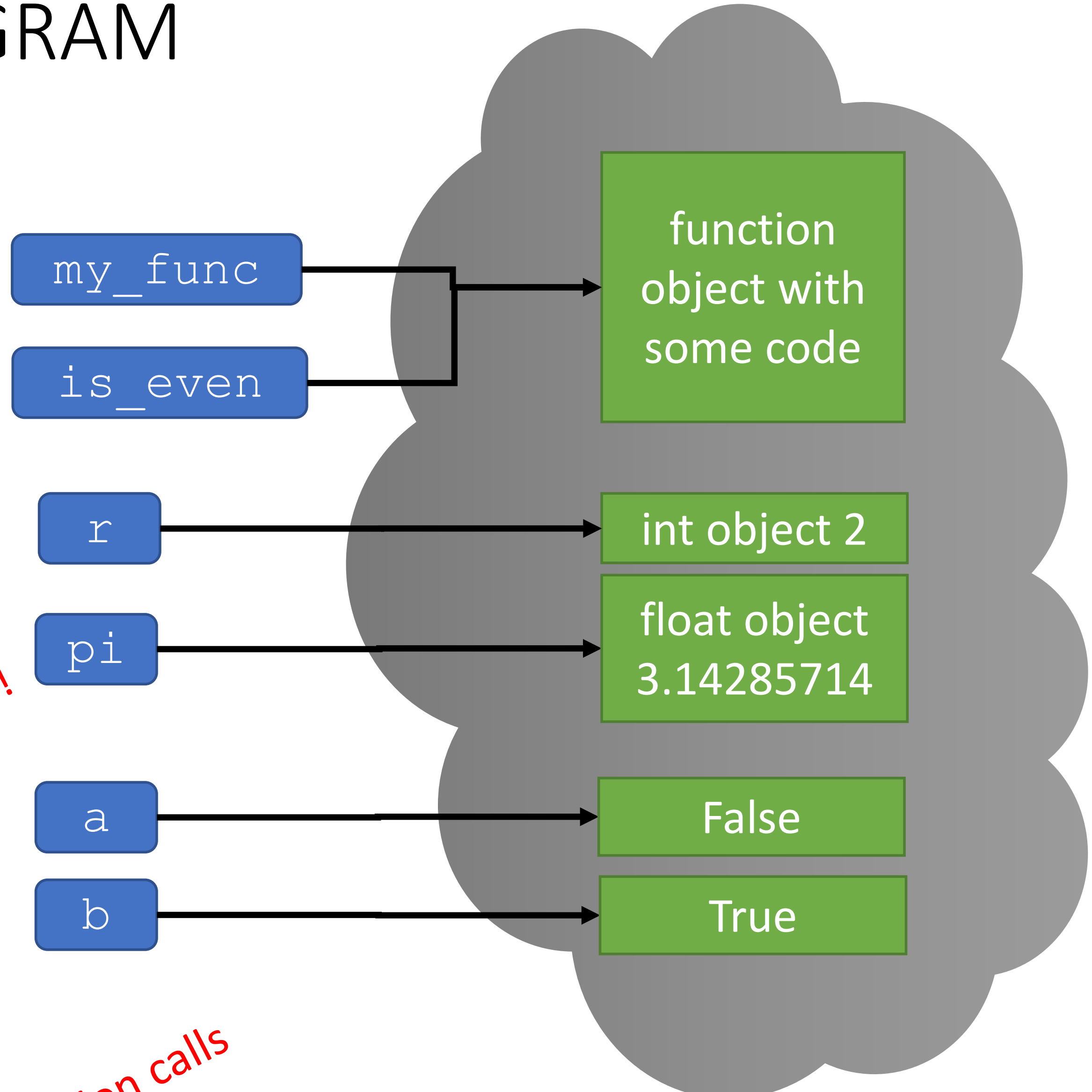
- Objects in Python have a type
  - int, float, str, Boolean, NoneType, function
- Objects can appear in RHS of assignment statement
  - Bind a name to an object
- Objects
  - Can be **used as an argument** to a procedure
  - Can be **returned as a value** from a procedure
- Functions are also **first class objects**!
- Treat functions just like the other types
  - Functions can be arguments to another function
  - Functions can be returned by another function

OBJECTS IN A PROGRAM

```python
def is_even(i):
    return i%2 == 0

r = 2

pi = 22/7

my_func = is_even

a = is_even(3)

b = my_func(4)
```

NOT a function call, just names!

Two function calls



BIG IDEA

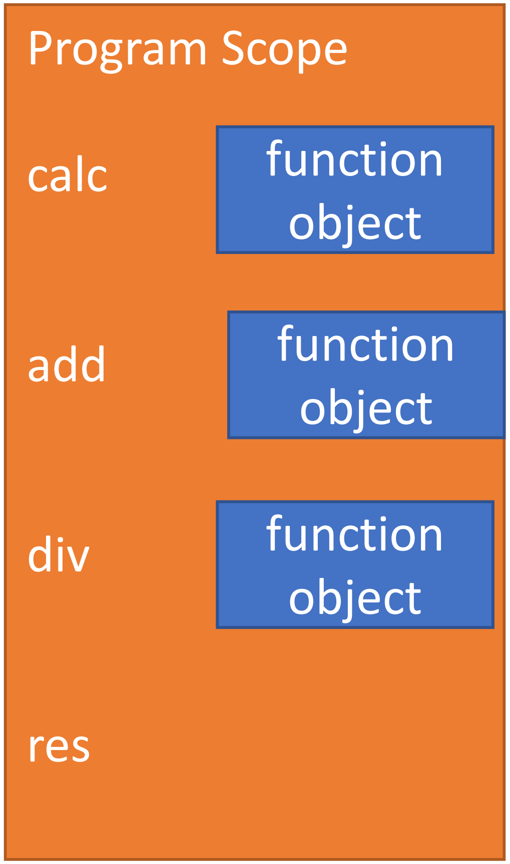
Everything in Python is an object.

FUNCTION AS A PARAMETER

```
def calc(op, x, y):
    return op(x,y)

def add(a,b):
    return a+b

def div(a,b):
    if b != 0:
        return a/b
    print("Denominator was 0.")

print(calc(add, 2, 3))
```

STEP THROUGH THE CODE

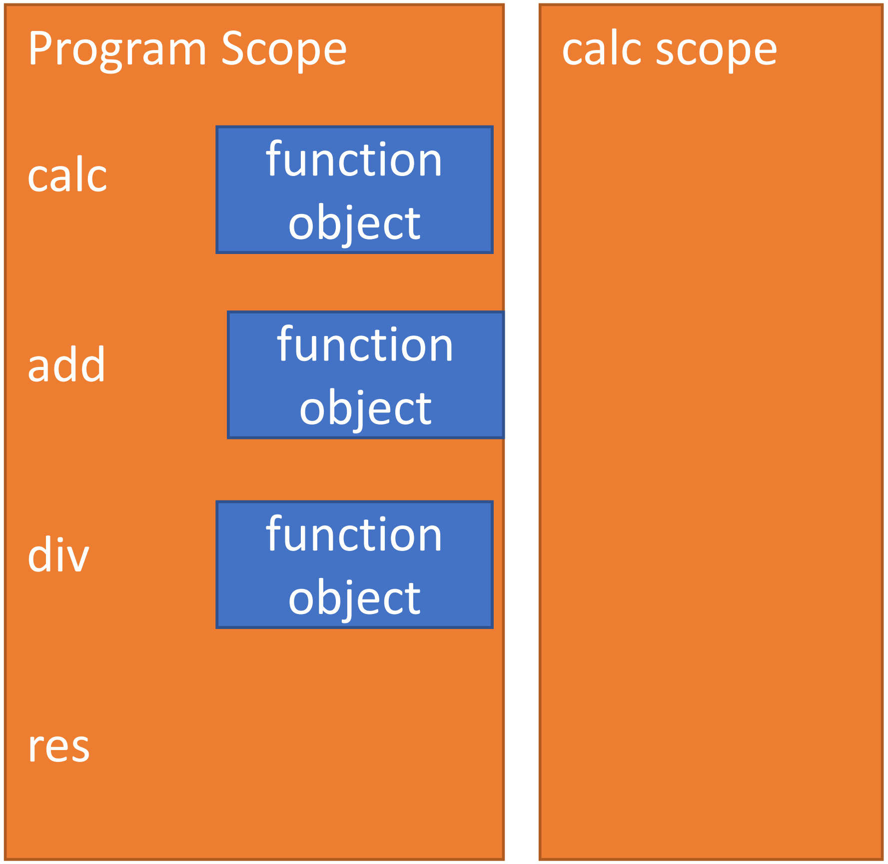
```
def calc(op, x, y):
    return op(x,y)

def add(a,b):
    return a+b

def div(a,b):
    if b != 0:
        return a/b
    print("Denom was 0.")

res = calc(add, 2, 3)
```



CREATE calc SCOPE

```
def calc(op, x, y):
    return op(x,y)

def add(a,b):
    return a+b

def div(a,b):
    if b != 0:
        return a/b
    print("Denom was 0.")

res = calc(add, 2, 3)
```

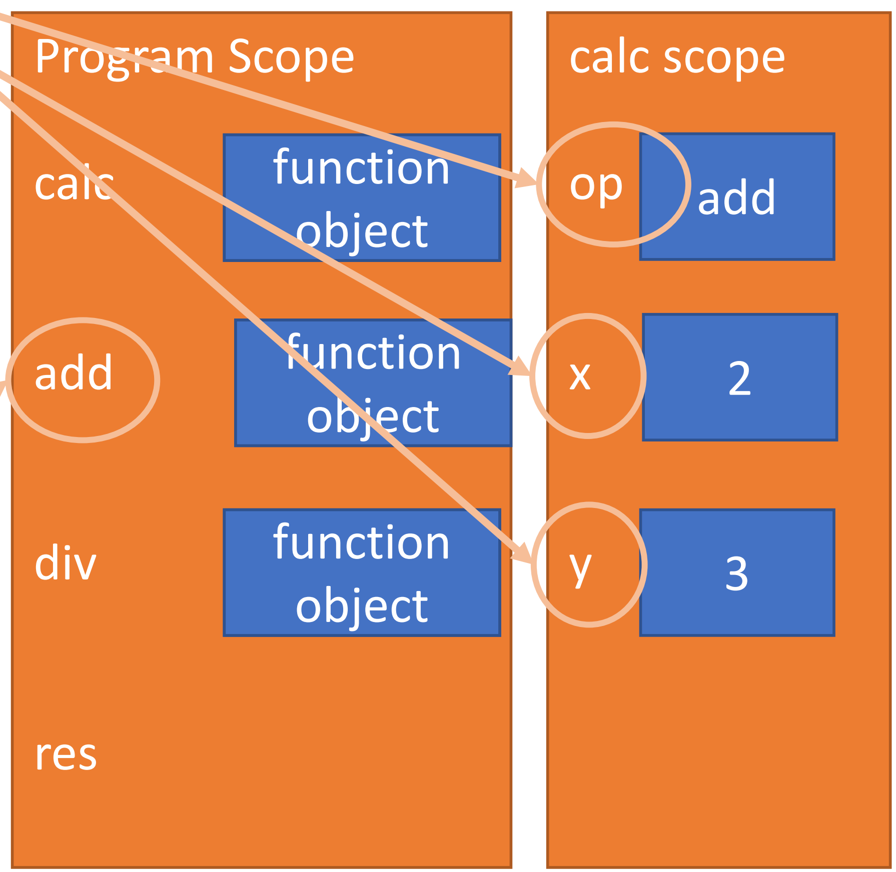
Function call



MATCH FORMAL PARAMS in calc

```
def calc(op, x, y):
    return op(x,y)

def add(a,b):
    return a+b

def div(a,b):
    if b != 0:
        return a/b
    print("Denom was 0.")

res = calc(add, 2, 3)
```

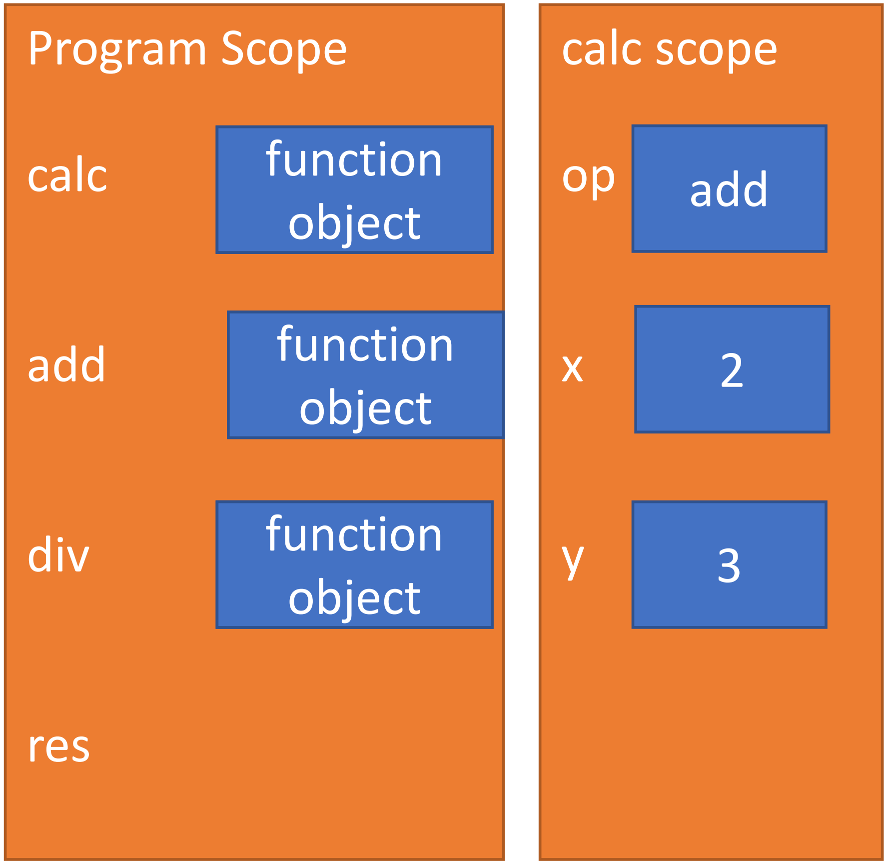
function obj NOT
function call



FIRST (and only) LINE IN calc

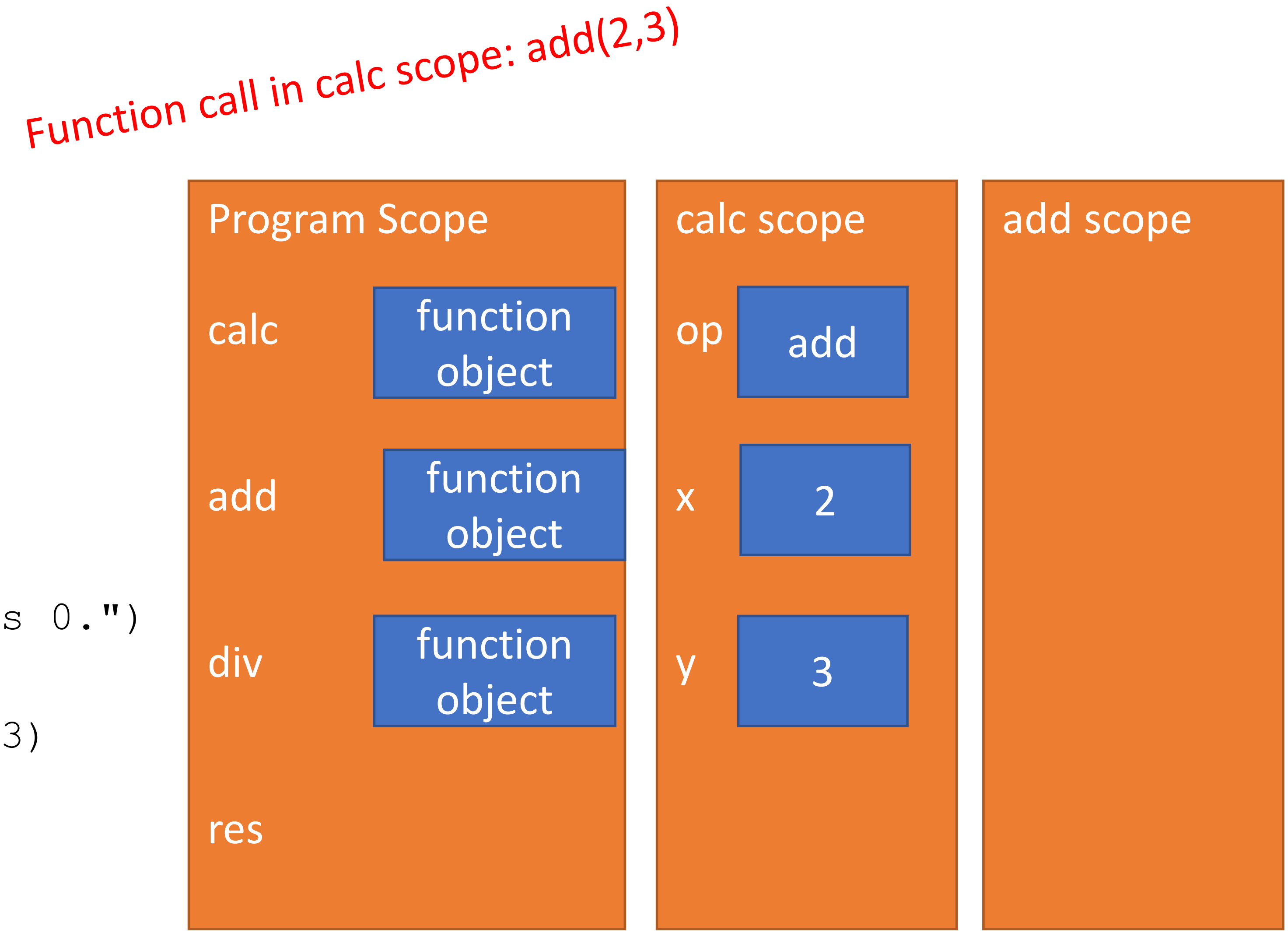
```
def calc(op, x, y):
    return op(x,y)

def add(a,b):
    return a+b

def div(a,b):
    if b != 0:
        return a/b
    print("Denom was 0.")

res = calc(add, 2, 3)
```

return add(2,3)
Just replace each param with its value



CREATE SCOPE OF add

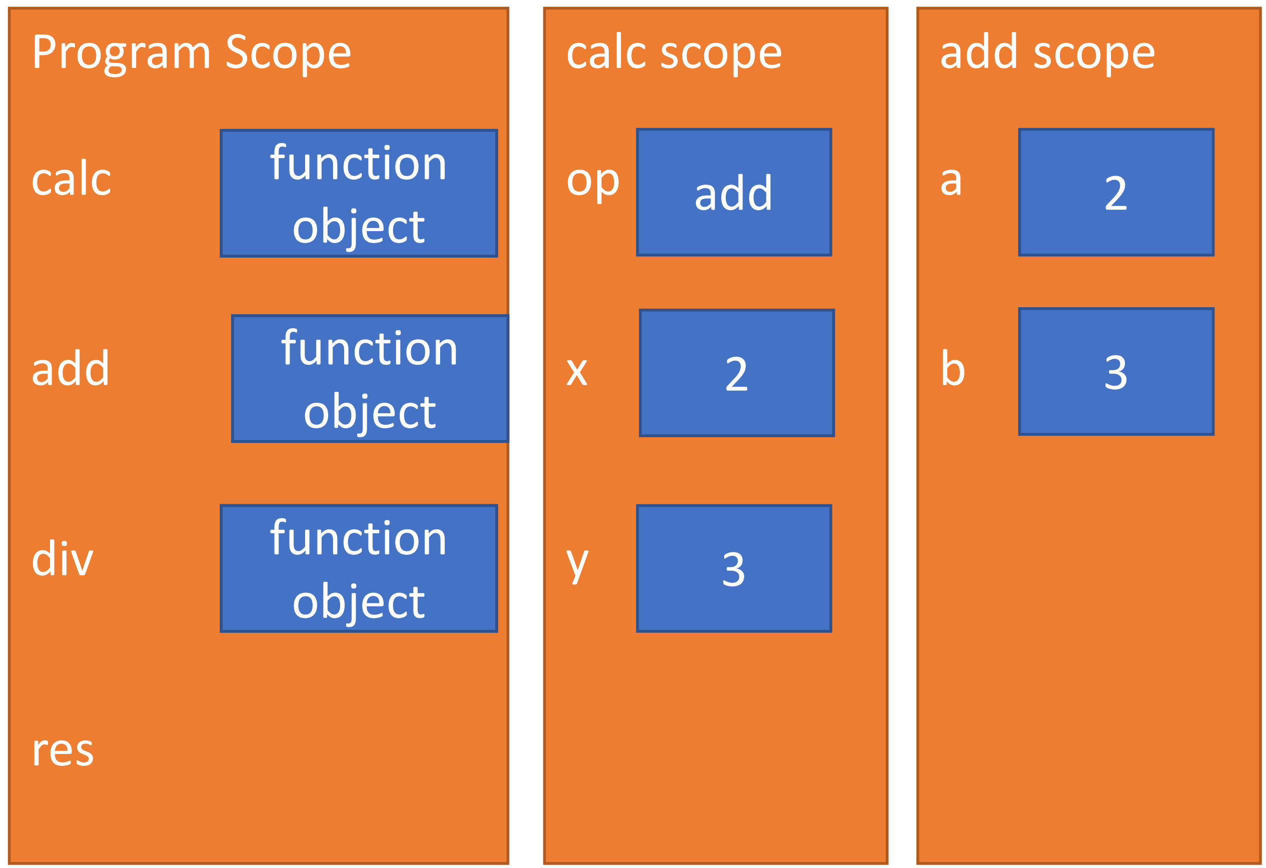
```
def calc(op, x, y):
    return op(x,y)

def add(a,b):
    return a+b

def div(a,b):
    if b != 0:
        return a/b
    print("Denom was 0.")

res = calc(add, 2, 3)
```



MATCH FORMAL PARAMS IN add

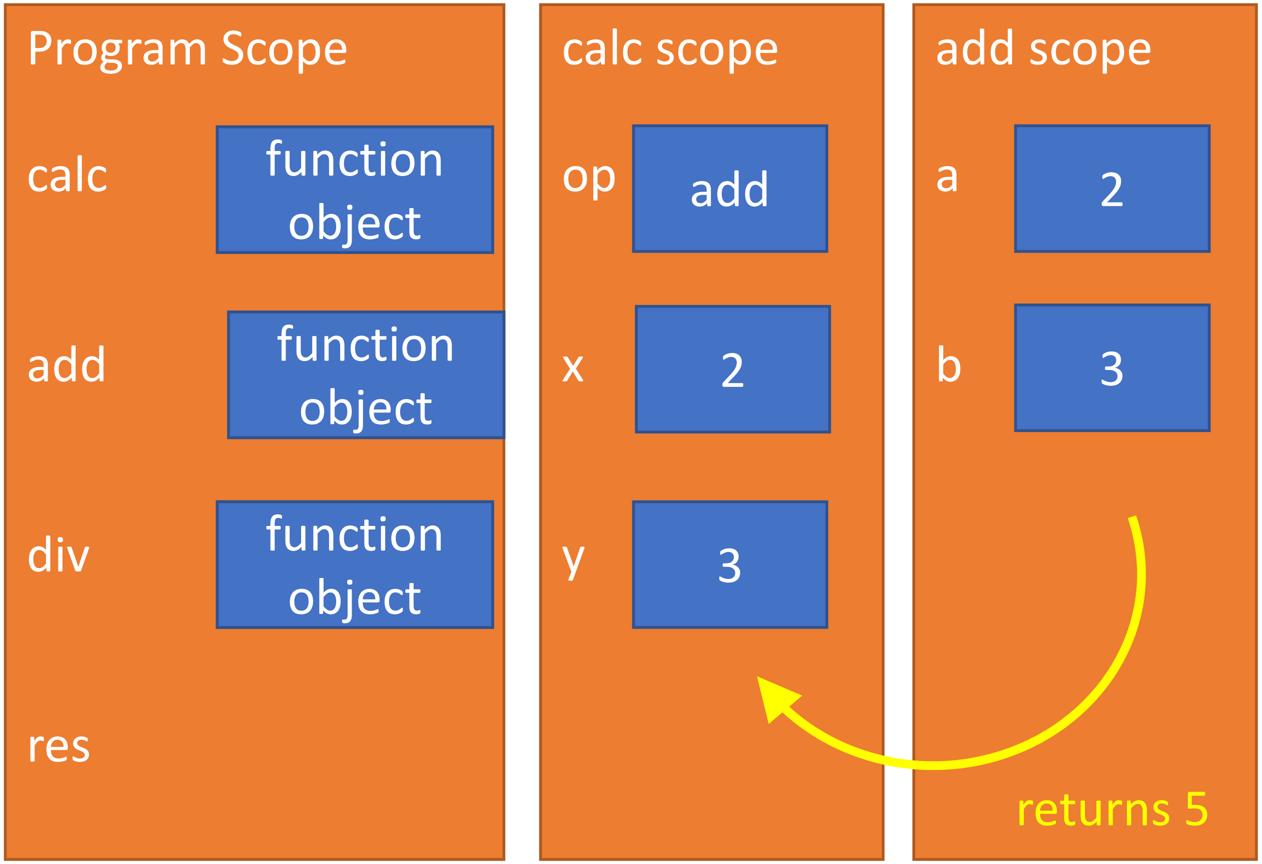
```
def calc(op, x, y):
    return op(x,y)

def add(a,b):
    return a+b

def div(a,b):
    if b != 0:
        return a/b
    print("Denom was 0.")

res = calc(add, 2, 3)
```

Function call in calc scope: add with formal params a=2 and b=3



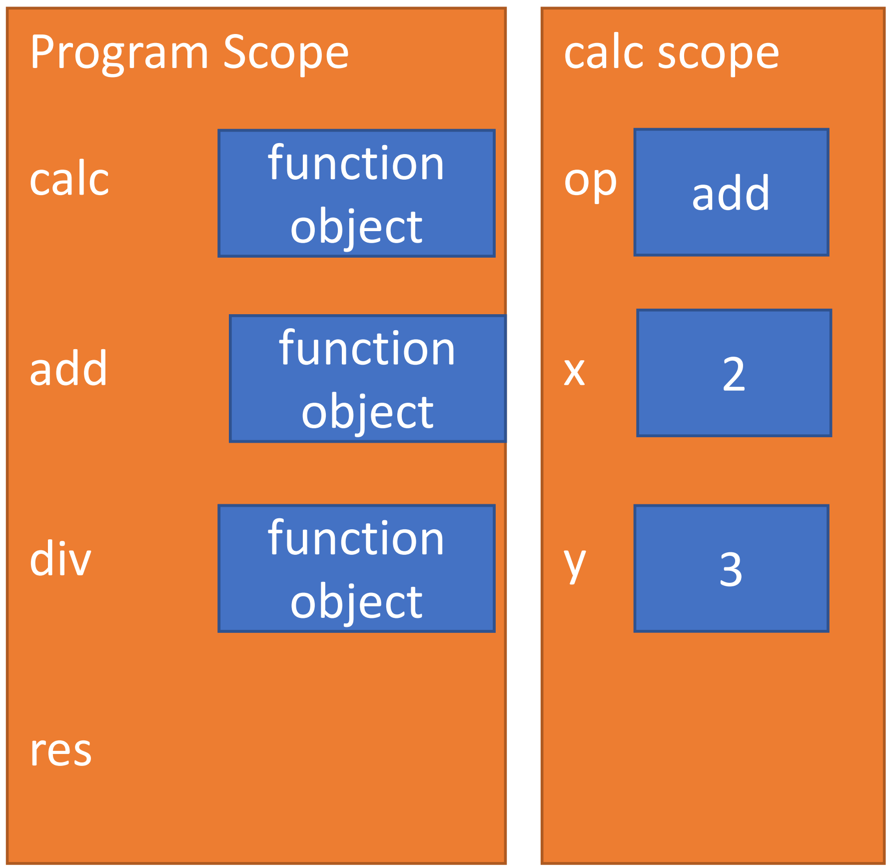
EXECUTE LINE OF add

```
def calc(op, x, y):
    return op(x,y)

def add(a,b):
    return a+b

def div(a,b):
    if b != 0:
        return a/b
    print("Denom was 0.")

res = calc(add, 2, 3)
```



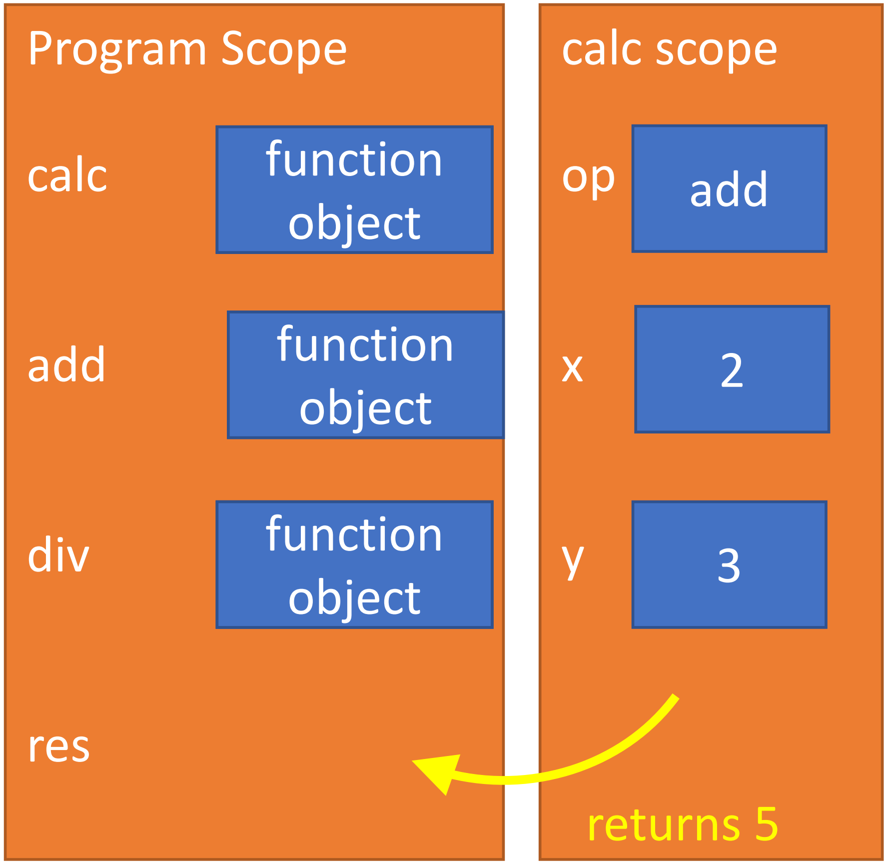
REPLACE FUNC CALL WITH RETURN

```
def calc(op, x, y):
    return op(x,y)   5

def add(a,b):
    return a+b

def div(a,b):
    if b != 0:
        return a/b
    print("Denom was 0.")

res = calc(add, 2, 3)
```



EXECUTE LINE OF calc

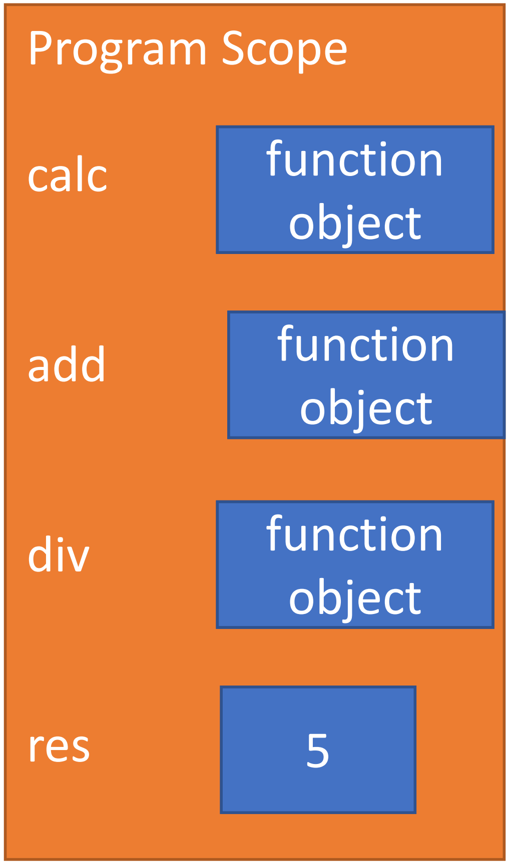
```
def calc(op, x, y):
    return op(x,y)

def add(a,b):
    return a+b

def div(a,b):
    if b != 0:
        return a/b
    print("Denom was 0.")

res = calc(add, 2, 3)
```



REPLACE FUNC CALL WITH RETURN

```
def calc(op, x, y):
    return op(x,y)

def add(a,b):
    return a+b

def div(a,b):
    if b != 0:
        return a/b
    print("Denom was 0.")

res = calc(add, 2, 3)
```

5



YOU TRY IT!

- Do a similar trace with the function call

```
def calc(op, x, y):
    return op(x,y)

def div(a,b):
    if b != 0:
        return a/b
    print("Denom was 0.")

res = calc(div,2,0)
```

What is the value of res and what gets printed?

ANOTHER EXAMPLE: FUNCTIONS AS PARAMS

```
def func_a():
    print('inside func_a')
def func_b(y):
    print('inside func_b')
    return y
def func_c(f, z):
    print('inside func_c')
    return f(z)
print(func_a())
print(5 + func_b(2))
print(func_c(func_b, 3))
```

call `func_a`, takes no parameters

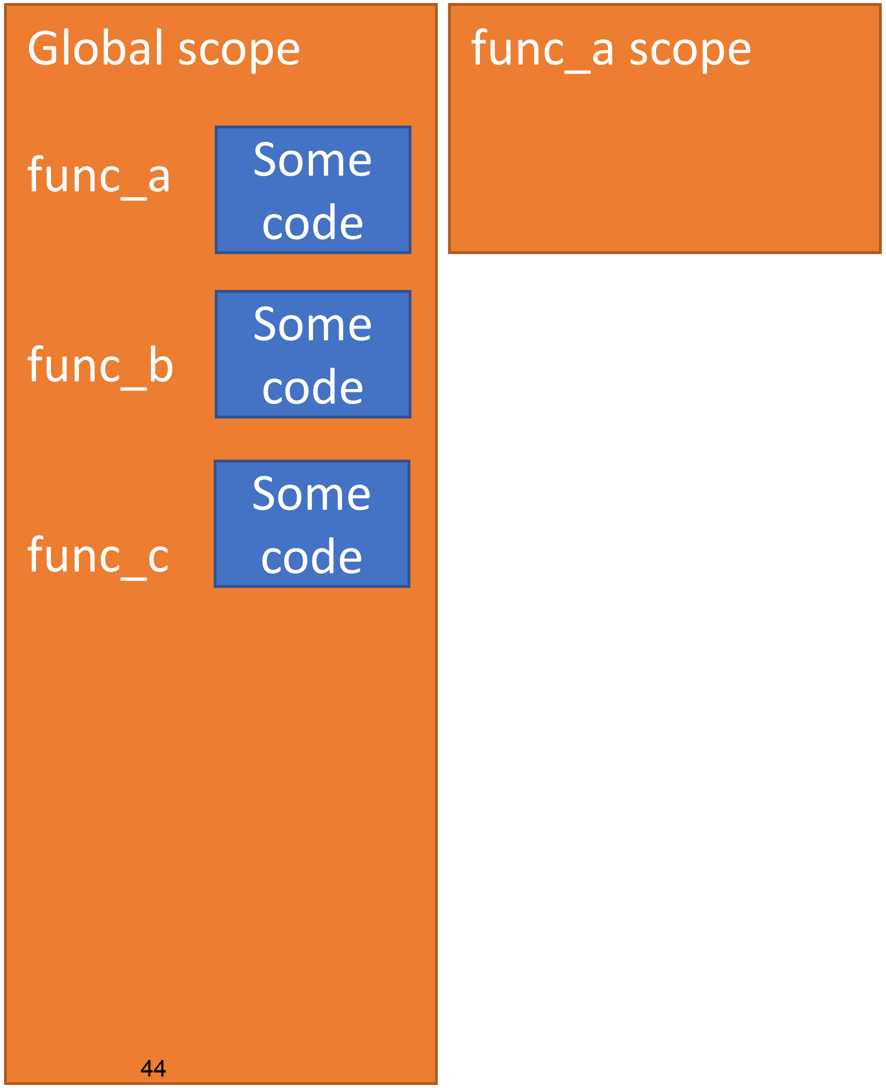
call `func_b`, takes one parameter, an int

call `func_c`, takes two parameters, another function and an int

FUNCTIONS AS PARAMETERS

```
def func_a():
    print('inside func_a')
def func_b(y):
    print('inside func_b')
    return y
def func_c(f, z):
    print('inside func_c')
    return f(z)
print(func_a())
print(5 + func_b(2))
print(func_c(func_b, 3))
```

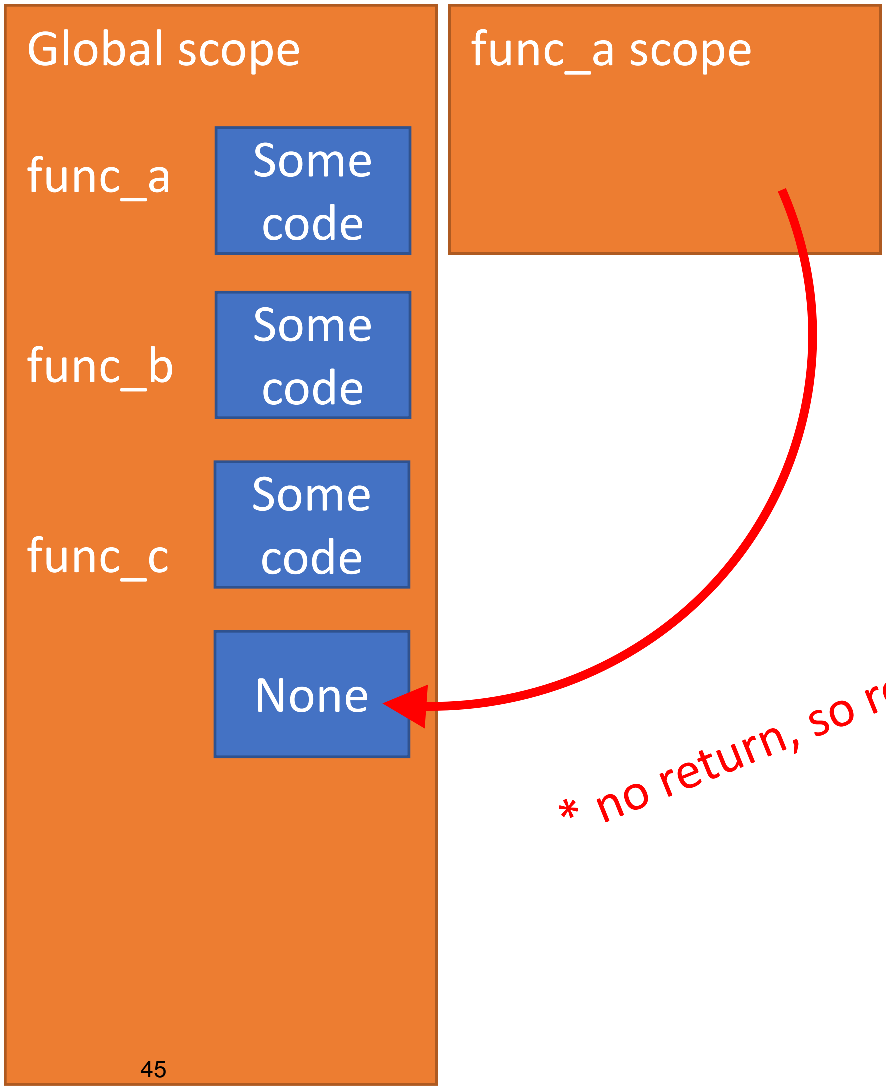
No bindings (no parameters)



FUNCTIONS AS PARAMETERS

body prints 'inside func_a' on console

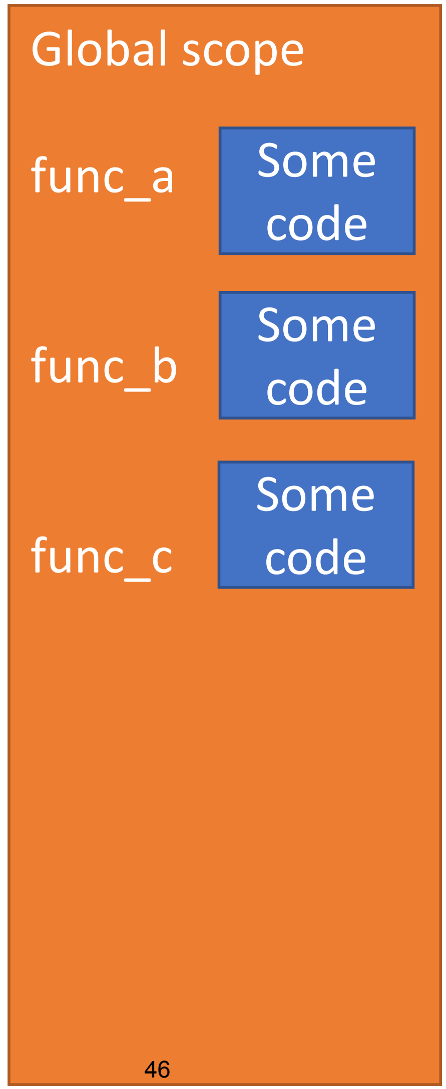
```
def func_a():
    print('inside func_a')
def func_b(y):
    print('inside func_b')
    return y
def func_c(f, z):
    print('inside func_c')
    return f(z)
print(func_a())
print(5 + func_b(2))
print(func_c(func_b, 3))
```



* no return, so returns None

FUNCTIONS AS PARAMETERS

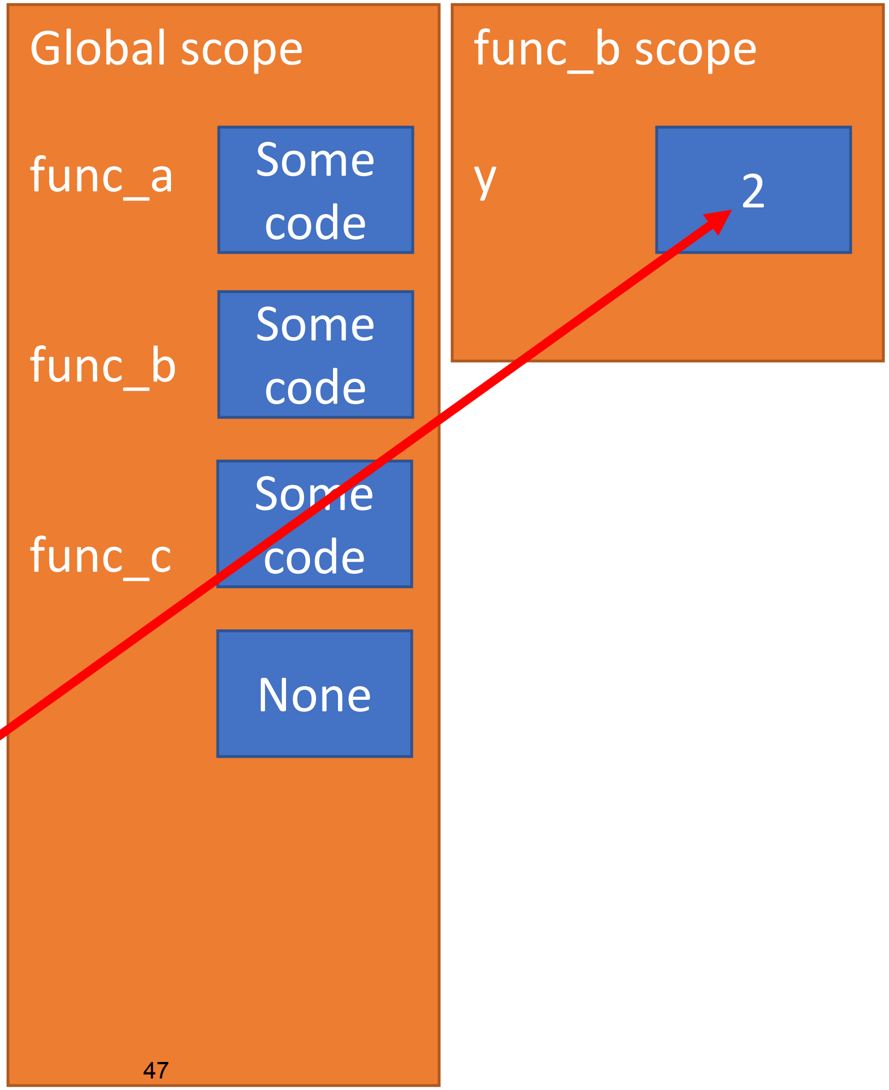
```
def func_a():
    print('inside func_a')
def func_b(y):
    print('inside func_b')
    return y
def func_c(f, z):
    print('inside func_c')
    return f(z)
print(func_a())
print(5 + func_b(2))
print(func_c(func_b, 3))
```



print displays None on console

FUNCTIONS AS PARAMETERS

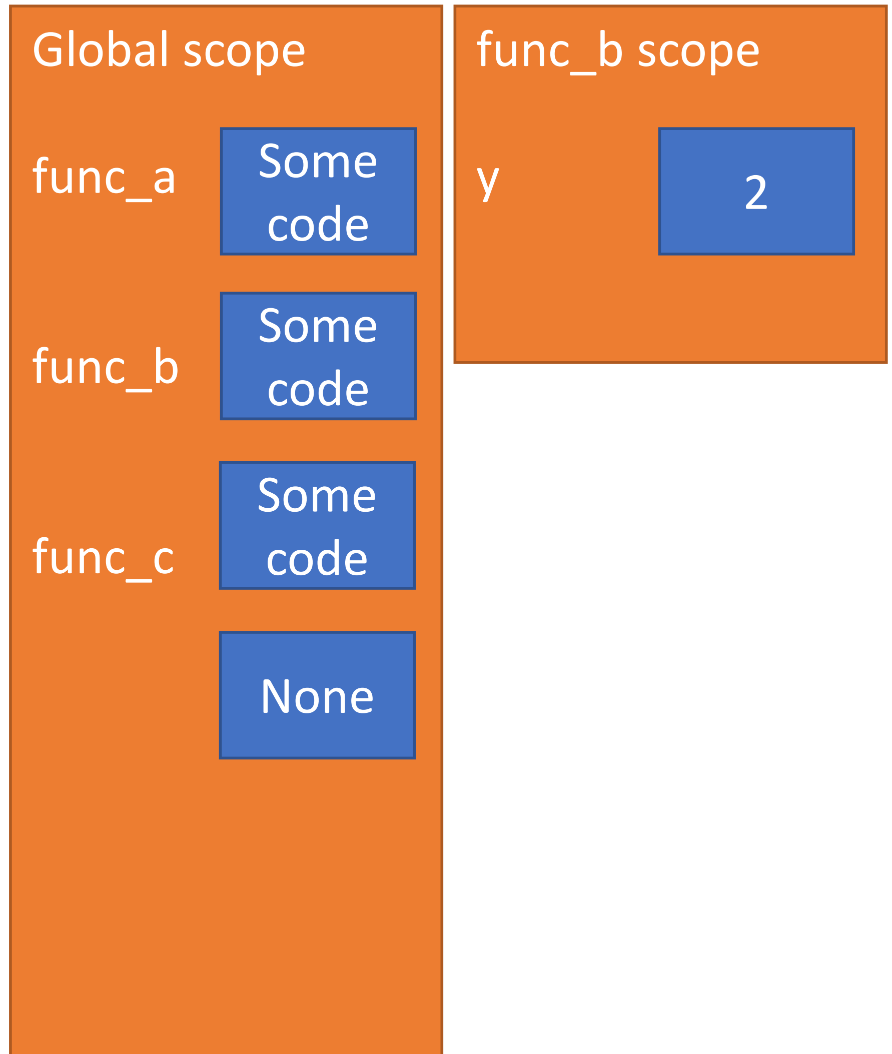
```
def func_a():
    print('inside func_a')
def func_b(y):
    print('inside func_b')
    return y
def func_c(f, z):
    print('inside func_c')
    return f(z)
print(func_a())
print(5 + func_b(2))
print(func_c(func_b, 3))
```



FUNCTIONS AS PARAMETERS

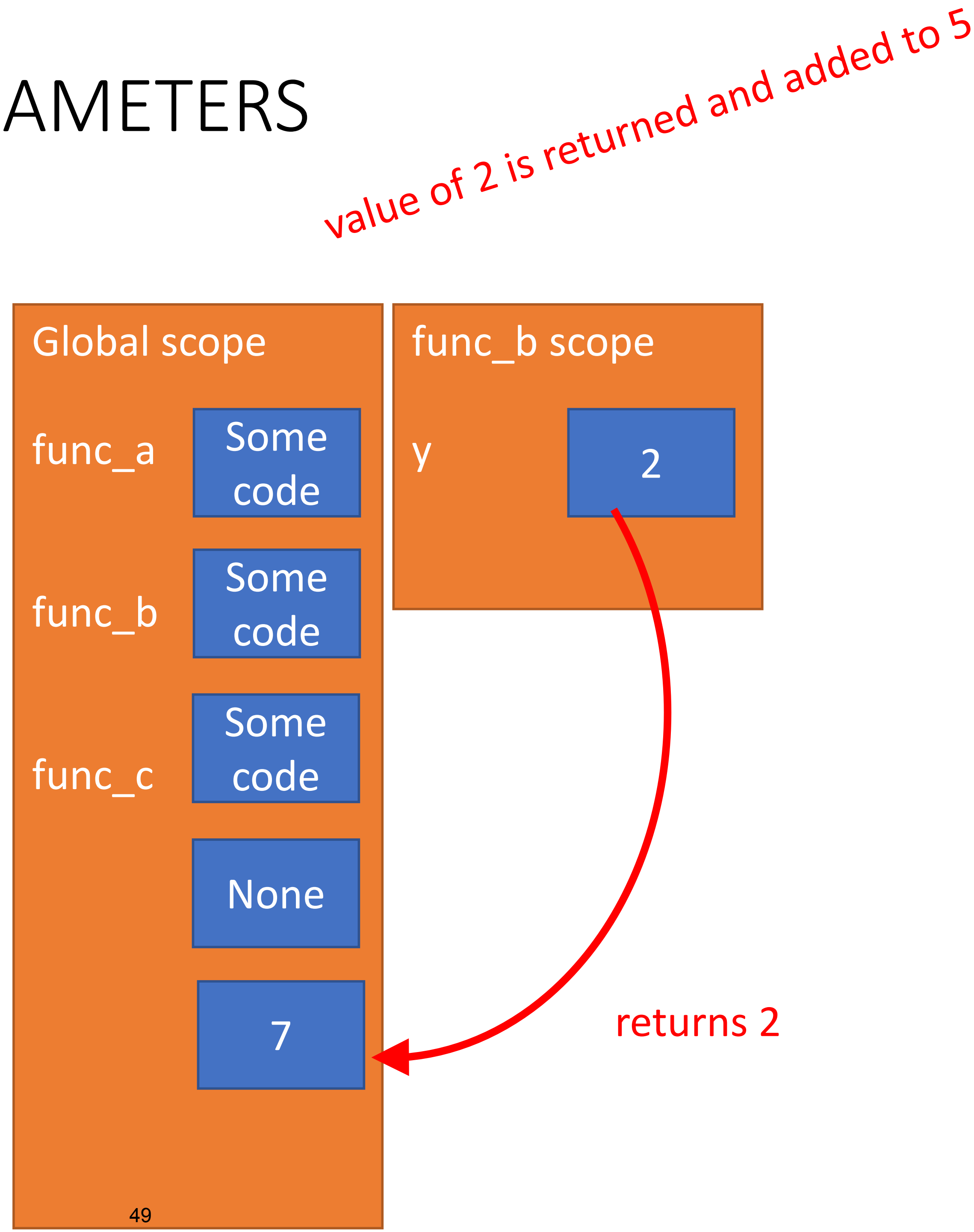
body prints 'inside func_b' on console

```
def func_a():
    print('inside func_a')
def func_b(y):
    print('inside func_b')
    return y
def func_c(f, z):
    print('inside func_c')
    return f(z)
print(func_a())
print(5 + func_b(2))
print(func_c(func_b, 3))
```



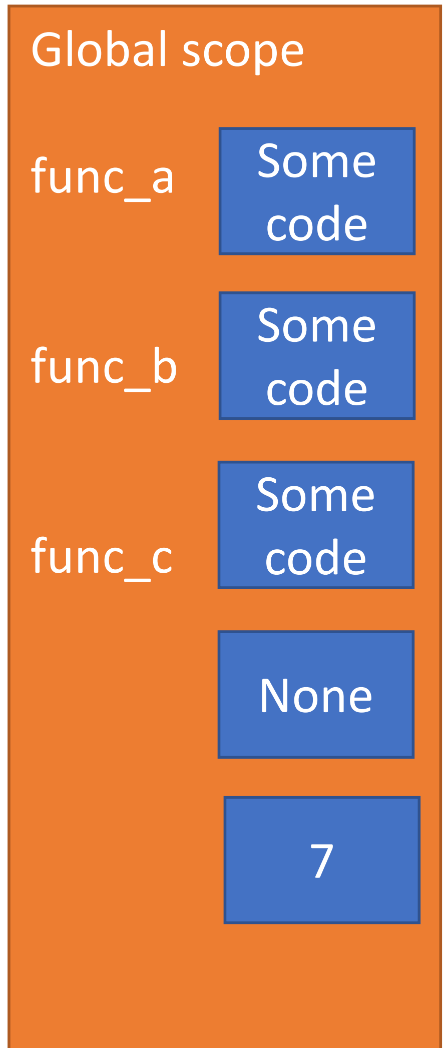
FUNCTIONS AS PARAMETERS

```
def func_a():
    print('inside func_a')
def func_b(y):
    print('inside func_b')
    return y
def func_c(f, z):
    print('inside func_c')
    return f(z)
print(func_a())
print(5 + func_b(2))
print(func_c(func_b, 3))
```



FUNCTIONS AS PARAMETERS

```
def func_a():
    print('inside func_a')
def func_b(y):
    print('inside func_b')
    return y
def func_c(f, z):
    print('inside func_c')
    return f(z)
print(func_a())
print(5 + func_b(2))
print(func_c(func_b, 3))
```

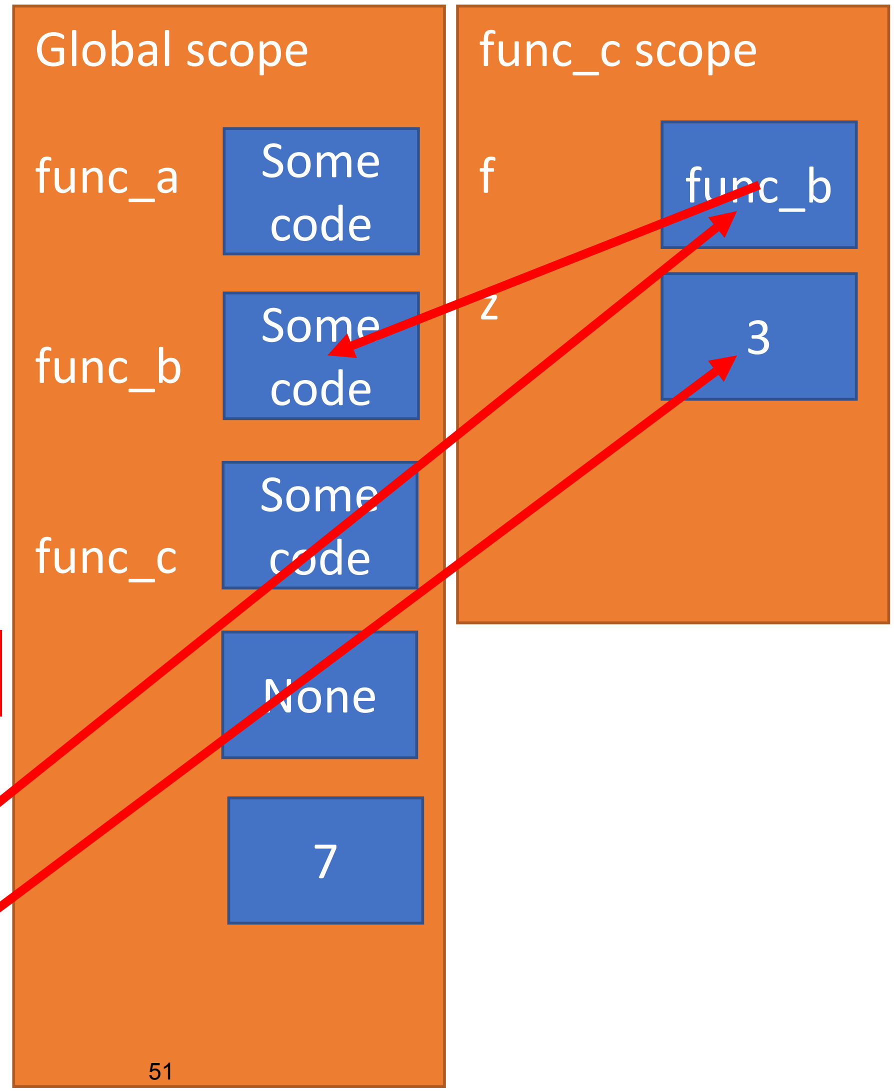


print displays 7 on console

FUNCTIONS AS PARAMETERS

body of func_c prints
'inside func_c' on console

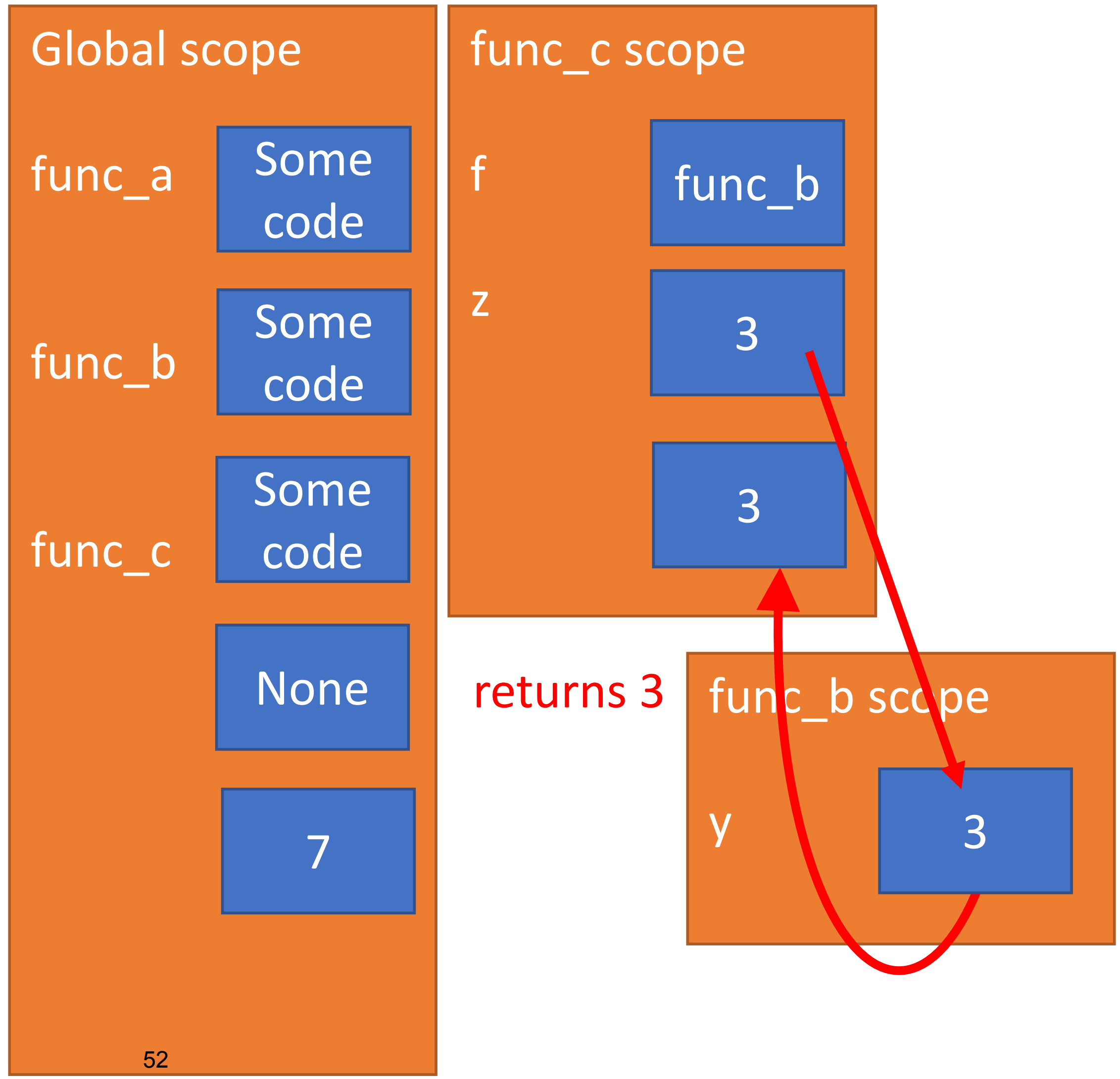
```
def func_a():
    print('inside func_a')
def func_b(y):
    print('inside func_b')
    return y
def func_c(f, z):
    print('inside func_c')
    return f(z)
print(func_a())
print(5 + func_b(2))
print(func_c(func_b, 3))
```



FUNCTIONS AS PARAMETERS

body of func_b
prints 'inside func_b'
on console

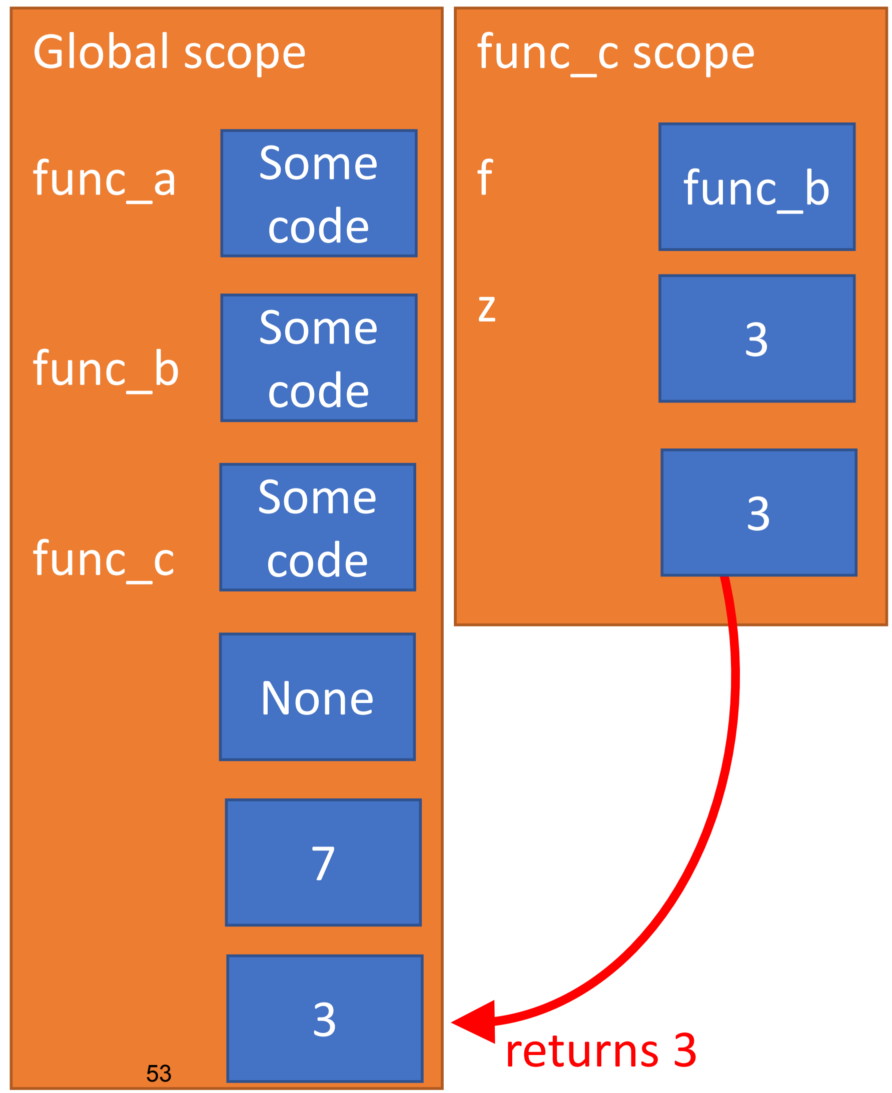
```
def func_a():
    print('inside func_a')
def func_b(y):
    print('inside func_b')
    return y
def func_c(f, z):
    print('inside func_c')
    return f(z)    3
print(func_a())
print(5 + func_b(2))
print(func_c(func_b, 3))
```



FUNCTIONS AS PARAMETERS

print displays 3 on console

```
def func_a():
    print('inside func_a')
def func_b(y):
    print('inside func_b')
    return y
def func_c(f, z):
    print('inside func_c')
    return f(z)
print(func_a())
print(5 + func_b(2))
print(func_c(func_b, 3))
```



YOU TRY IT!

- Write a function that meets these specs.

```
def apply(criteria,n):
    """
    * criteria is a func that takes in a number and returns a bool
    * n is an int
    Returns how many ints from 0 to n (inclusive) match
    the criteria (i.e. return True when run with criteria)
    """
```

SUMMARY

- Functions are first class objects
  - They have a **type**
  - They can be **assigned as a value** bound to a name
  - They can be used as an **argument** to another procedure
  - They can be **returned** as a value from another procedure
- Have to be careful about environments
  - Main program runs in the global environment
  - Function calls each get a new temporary environment
- This enables the creation of concise, easily read code

LAMBDA FUNCTIONS, TUPLES and LISTS

(download slides and .py files to follow along)

6.100L Lecture 9

Ana Bell

FROM LAST TIME

```
def apply(criteria,n):
    """
    * criteria: function that takes in a number and returns a bool
    * n: an int
    Returns how many ints from 0 to n (inclusive) match the
    criteria (i.e. return True when run with criteria) """
    count = 0
    for i in range(n+1):
        if criteria(i):
            count += 1
    return count

def is_even(x):
    return x%2==0

print(apply(is_even,10))
```

ANONYMOUS FUNCTIONS

- Sometimes don't want to name functions, especially simple ones. This function is a good example:

```
def is_even(x):
    return x%2==0
```

- Can use an **anonymous** procedure by using `lambda`

```
lambda x: x%2 == 0
```

parameter

Body of lambda
Note no return keyword

- `lambda` creates a procedure/function object, but simply does not bind a name to it

ANONYMOUS FUNCTIONS

- Function call with a named function:

```
apply( is_even , 10 )
```

- Function call with an anonymous function as parameter:

```
apply( lambda x: x%2 == 0 , 10 )
```

- `lambda` function is **one-time use**. It can’t be reused because it has no name!

YOU TRY IT!

- What does this print?

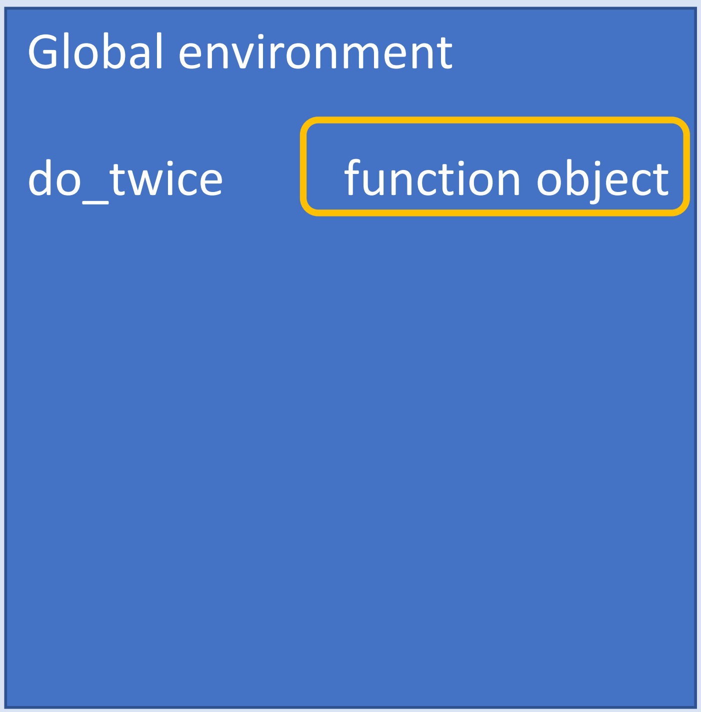
```
def do_twice(n, fn):
    return fn(fn(n))

print(do_twice(3, lambda x: x**2))
```

YOU TRY IT!

- What does this print?

```
def do_twice(n, fn):
    return fn(fn(n))
```

```
print(do_twice(3, lambda x: x**2))
```



YOU TRY IT!

- What does this print?

```
def do_twice(n, fn):
    return fn(fn(n))

print(do_twice(3, lambda x: x**2))
```

Global environment

do_twice	function object

do_twice environment

n	3
fn	`lambda x: x**2`

YOU TRY IT!

- What does this print?

```
def do_twice(n, fn):
    return fn(fn(n))

print(do_twice(3, lambda x: x**2))
```

Global environment

do_twice function object

do_twice environment

n 3
fn `lambda x: x**2`

`lambda x: x**2` environment

x ???

YOU TRY IT!

- What does this print?

```
def do_twice(n, fn):
    return fn(fn(n))

print(do_twice(3, lambda x: x**2))
```

Global environment

do_twice   function object

do_twice environment

n   3
fn   `lambda x: x**2`

`lambda x: x**2` environment

x   ???

`lambda x: x**2` environment

x   3

YOU TRY IT!

- What does this print?

```
def do_twice(n, fn):
    return fn(fn(n))

print(do_twice(3, lambda x: x**2))
```

9

Global environment

do_twice   function object

do_twice environment

n   3
fn   `lambda x: x**2`

`lambda x: x**2` environment

x   **9**

`lambda x: x**2` environment

x   3

Returns 9

YOU TRY IT!

- What does this print?

```
def do_twice(n, fn):
    return fn(fn(n))

print(do_twice(3, lambda x: x**2))
```

81

Global environment

do_twice function object

do_twice environment

n 3

fn `lambda x: x**2`

`lambda x: x**2` environment

x **9**

Returns 81

YOU TRY IT!

- What does this print?

```
def do_twice(n, fn):
    return fn(fn(n))

print(do_twice(3, lambda x: x**2))
```

81

Global environment

do_twice   function object

PRINTS 81

do_twice environment

n   3
fn   lambda x: x**2

Returns 81

TUPLES

A NEW DATA TYPE

- Have seen scalar types: `int, float, bool`
- Have seen one compound type: `string`
- Want to introduce more general **compound data types**
  - Indexed sequences of elements, which could themselves be compound structures
  - **Tuples** – immutable
  - **Lists** – mutable

- Next lecture, will explore ideas of
  - Mutability
  - Aliasing
  - Cloning

TUPLES

Remember strings?

- **Indexable ordered sequence** of objects
  - Objects can be any type – int, string, tuple, tuple of tuples, …
- Cannot change element values, **immutable**

```
te = ()
```
Empty tuple

```
ts = (2,)
```
Extra comma means tuple with one element

Compare with `ts = (2)`

```
t = (2, "mit", 3)
```
Multiple elements in tuple separated by commas

`t[0]` → evaluates to 2 Indexing starts at 0

`(2,"mit",3) + (5,6)` → evaluates to a new tuple `(2,"mit",3,5,6)`

`t[1:2]` → slice tuple, evaluates to `("mit",)`

`t[1:3]` → slice tuple, evaluates to `("mit",3)`

`len(t)` → evaluates to 3

`max((3,5,0))` → evaluates 5 Other functions also work, e.g, sum

`t[1] = 4` → gives error, **can't modify object**

INDICES AND SLICING

Remember strings?

```
seq = (2,'a',4,(1,2))
  index:  0   1   2    3
print(len(seq))          → 4
print(seq[3])            → (1,2)
print(seq[-1])           → (1,2)
print(seq[3][0])         → 1
print(seq[4])            → error
```

An element of a sequence is at an **index**, indices start at 0

```
print(seq[1])            → 'a'
print(seq[-2:]           → (4,(1,2))
print(seq[1:4:2]         → ('a',(1,2))
print(seq[:-1])          → (2,'a',4)
print(seq[1:3])          → ('a',4)
```

Slices extract subsequences. Indices evaluated from left to right

```
for e in seq:            → 2
    print(e)               a
                           4
                          (1,2)
```

Iterating over sequences

TUPLES

- Conveniently used to **swap** variable values

```
x = 1
y = 2
x = y
y = x
```

```
x = 1
y = 2
temp = x
x = y
y = temp
```

```
x = 1
y = 2
(x, y) = (y, x)
```

First, create tuple
y = 2
x = 1

Then, bind values in new tuple

TUPLES

- Used to **return more than one value** from a function

```
def quotient_and_remainder(x, y):
    q = x // y
    r = x % y
    return (q, r)
```

One object!
(with 2 elements/values)

(3,1)

(3, 1)

```
both = quotient_and_remainder(10,3)
```

1

(2, 1)

2

```
(quot, rem) = quotient_and_remainder(5,2)
```

BIG IDEA

Returning
one object (a tuple)
allows you to return
multiple values (tuple elements)

YOU TRY IT!

- Write a function that meets these specs:
- Hint: remember how to check if a character is in a string?

```
def char_counts(s):
    """ s is a string of lowercase chars
    Return a tuple where the first element is the
    number of vowels in s and the second element
    is the number of consonants in s """
```

VARIABLE NUMBER of ARGUMENTS

- Python has some built-in functions that take variable number of arguments, e.g, `min`
- Python allows a programmer to have same capability, using *** notation**

```
def mean(*args):
    tot = 0
    for a in args:
        tot += a
    return tot/len(args)
```

- `numbers` is bound to a **tuple of the supplied values**
- Example:
  - `mean(1,2,3,4,5,6)`   args → (1,2,3,4,5,6)

LISTS

LISTS

- **Indexable ordered sequence** of objects
  - Usually homogeneous (i.e., all integers, all strings, all lists)
  - But can contain mixed types (not common)
- Denoted by **square brackets**, `[]`

Tuples were ()

- **Mutable**, this means you can change values of specific elements of list

Remember tuples are immutable – you **cannot** change element values.
Lists are mutable, you can change them directly.

INDICES and ORDERING

Remember strings and tuples?

```
a_list = []
```

empty list

```
L = [2, 'a', 4, [1,2]]
```

`[1,2]+[3,4]` → evaluates to `[1,2,3,4]`

`len(L)` → evaluates to 4

Gives length of top level of tuple

`L[0]` → evaluates to 2

Indexing starts at 0

`L[2]+1` → evaluates to 5

`L[3]` → evaluates to `[1,2]`, another list!

`L[4]` → gives an error

```
i = 2
```

`L[i-1]` → evaluates to `'a'` since `L[1]='a'`

`max([3,5,0])` → evaluates 5

ITERATING OVER a LIST

- Compute the **sum of elements** of a list
- Common pattern

```
total = 0
for i in range(len(L)):
    total += L[i]
print(total)
```

```
total = 0
for i in L:
    total += i
print(total)
```

Like strings, can iterate over elements of list **directly**

- Notice
  - list elements are indexed `0` to `len(L)-1` and `range(n)` goes from `0` to `n-1`

This version is more "pythonic"!

ITERATING OVER a LIST

- Natural to capture iteration over a list inside a function

```
total = 0
for i in L:
    total += i
print(total)
```

```
def list_sum(L):
    total = 0
    for i in L:
        # i is 8 then 3 then 5
        total += i
    return total
```

- Function call `list_sum([8,3,5])`
  - **Loop variable `i` takes on values in the list in order!** 8 then 3 then 5
  - To help you write code and debug, comment on what the loop var values are so you don't get confused!

LISTS SUPPORT ITERATION

- Because lists are ordered sequences of elements, they naturally interface with iterative functions

Add the ***elements*** of a list

```
def list_sum(L):
    total = 0
    for e in L:
        total += e
    return(total)
```

e is:
1 then 3 then 5

```
list_sum([1,3,5]) → 9
```

Add the ***length of elements*** of a list

```
def len_sum(L):
    total = 0
    for s in L:
        total += len(s)
    return(total)
```

s is:
'ab' then 'def' then 'g'

2 then 3 then 1

```
len_sum(['ab', 'def', 'g']) → 6
```

YOU TRY IT!

- Write a function that meets these specs:

```
def sum_and_prod(L):
    """ L is a list of numbers
    Return a tuple where the first value is the
    sum of all elements in L and the second value
    is the product of all elements in L """
```

SUMMARY

- **Lambda functions** are useful when you need a simple function once, and whose body can be written in one line
- **Tuples** are indexable sequences of objects
  - Can’t change its elements, for ex. can’t add more objects to a tuple
  - Syntax is to use ()
- **Lists** are indexable sequences of objects
  - Can change its elements. Will see this next time!
  - Syntax is to use []
- Lists and tuples are very **similar to strings** in terms of
  - Indexing,
  - Slicing,
  - Looping over elements

6.100L Introduction to Computer Science and Programming Using Python
Fall 2022

LISTS, MUTABILITY

(download slides and .py files to follow along)

6.100L Lecture 10

Ana Bell

INDICES and ORDERING in LISTS

`a_list =` $\boxed{\texttt{[]}}$ empty list

`L = [2, 'a', 4, [1,2]]`

`len(L)` → evaluates to 4

`L[0]` → evaluates to 2 Indexing starts at 0

`L[3]` → evaluates to `[1,2]`, another list!

`[2,'a'] + [5,6]` → evaluates to `[2,'a',5,6]`

`max([3,5,0])` → evaluates to 5

`L[1:3]` → evaluates to `['a', 4]` Slicing just like strings

`for e in L` → loop variable becomes each element in L

`L[3] = 10` → **mutates `L` to `[2,'a',4,10]`**

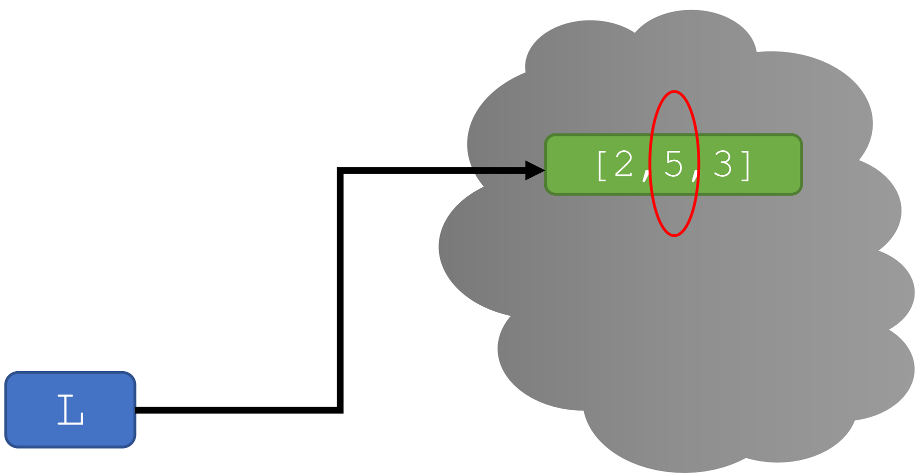
Mutate L by changing an element

MUTABILITY

- Lists are **mutable**!
- Assigning to an element at an index **changes** the value

```
L = [2, 4, 3]
L[1] = 5
```

- `L` is now `[2, 5, 3]`; note this is the **same object** `L`

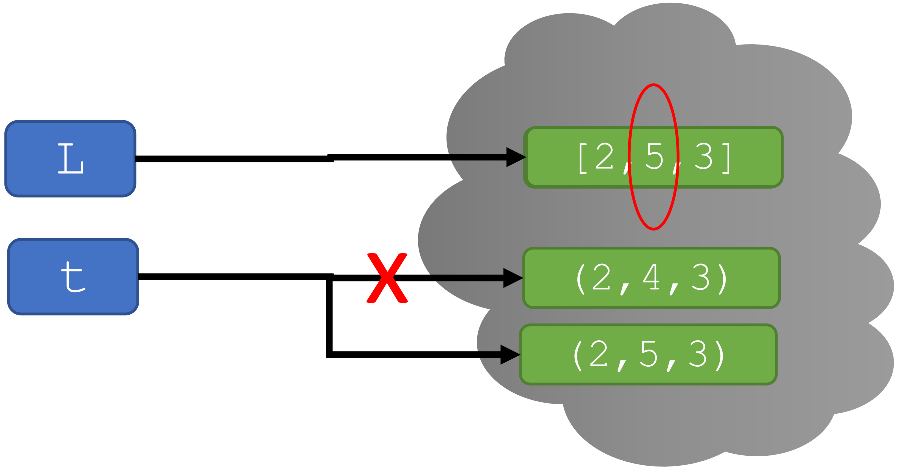


different from strings and tuples!

MUTABILITY

- Compare
  - Making L by **mutating an element** vs.
  - Making t by **creating a new object**

```
L = [2, 4, 3]
L[1] = 5
t = (2, 4, 3)
t = (2, 5, 3)
```



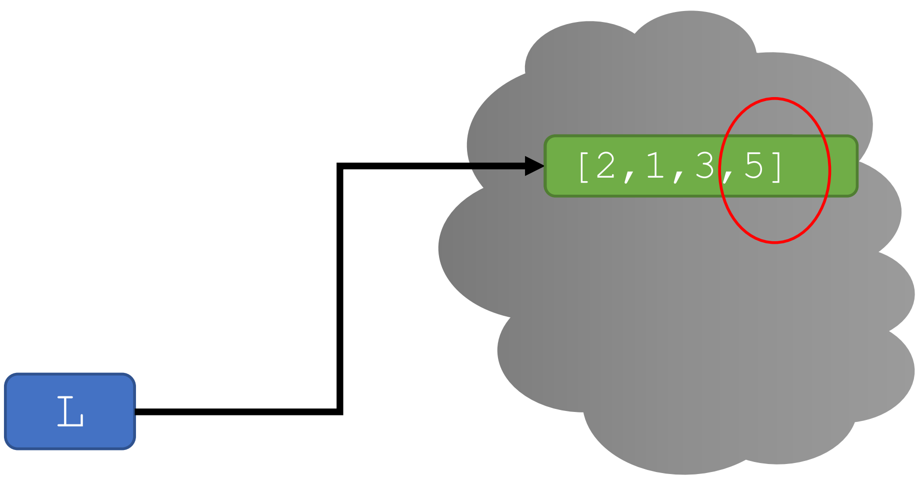
OPERATION ON LISTS – append

Function name

- **Add** an element to end of list with `L.append(element)`
- **Mutates** the list!

```
L = [2,1,3]
L.append(5)
```
→ L is now `[2,1,3,5]`

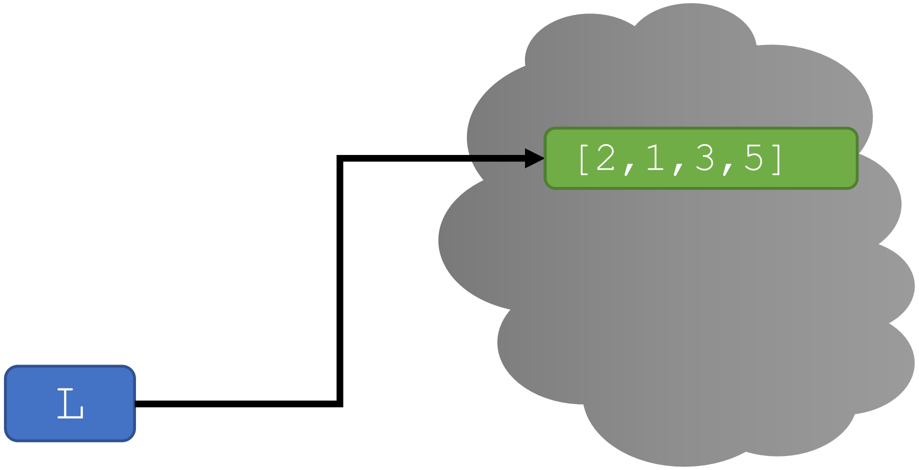
L and element are your objects



OPERATION ON LISTS – append

- **Add** an element to end of list with `L.append(element)`
- **Mutates** the list!

```
L = [2,1,3]
L.append(5)
```
→ L is now `[2,1,3,5]`
```
L = L.append(5)
```

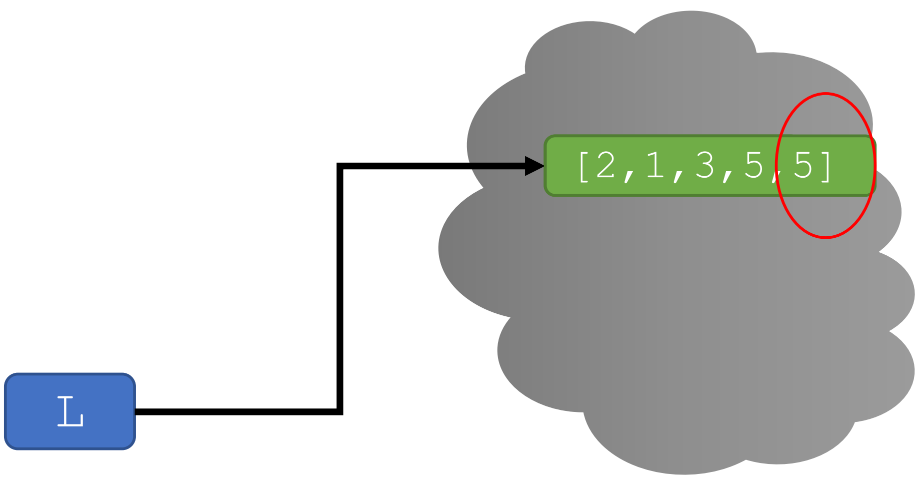


OPERATION ON LISTS – append

- **Add** an element to end of list with `L.append(element)`
- **Mutates** the list!

```
L = [2,1,3]
L.append(5)
```
→ L is now `[2,1,3,5]`

```
L = L.append(5)
```



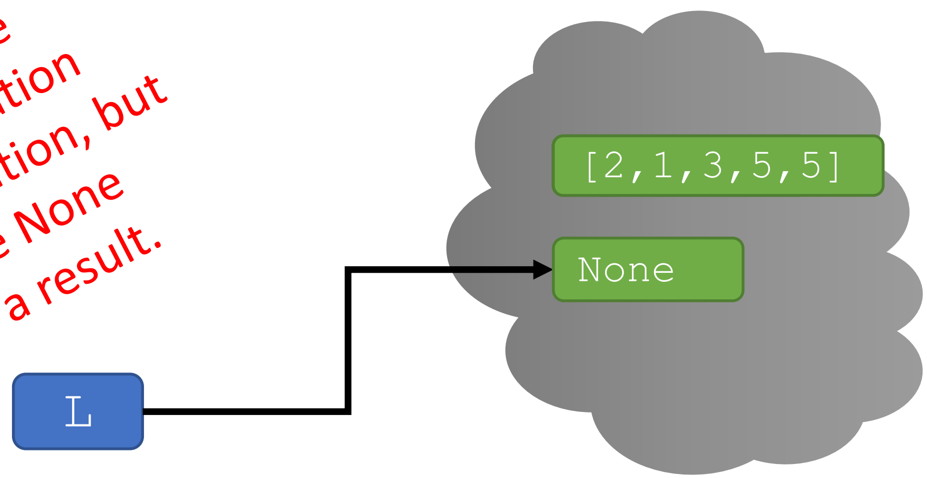
OPERATION ON LISTS – append

- **Add** an element to end of list with `L.append(element)`
- **Mutates** the list!

```
L = [2,1,3]
L.append(5)
```
→ L is now `[2,1,3,5]`

$\boxed{\texttt{L = L.append(5)}}$

Be careful! The append operation does a mutation, but returns the None object as a result.

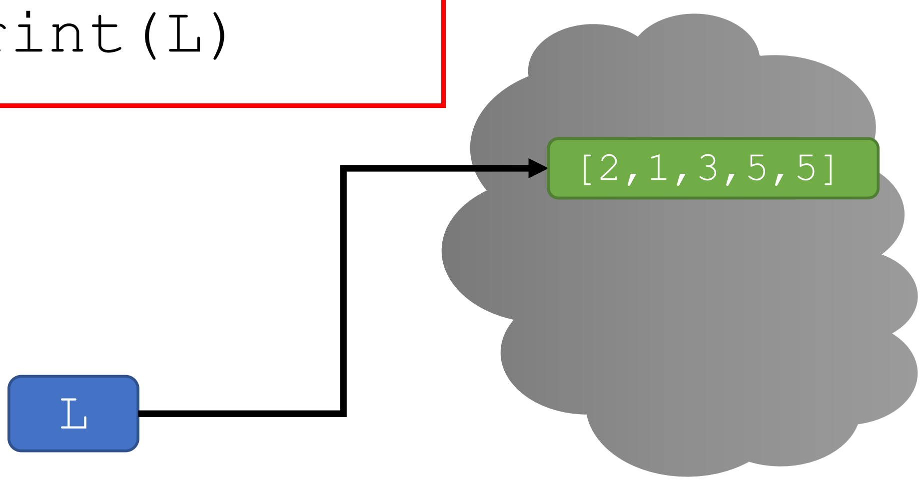


OPERATION ON LISTS – append

- **Add** an element to end of list with `L.append(element)`
- **Mutates** the list!

```
L = [2,1,3]
L.append(5)
```
→ L is now `[2,1,3,5]`

```
L.append(5)
print(L)
```
→ L is now `[2,1,3,5,5]`



Append is used strictly for its **side effect**

YOU TRY IT!

- What is the value of L1, L2, L3 and L at the end?

```
L1 = ['re']
L2 = ['mi']
L3 = ['do']
L4 = L1 + L2
L3.append(L4)
L = L1.append(L3)
```

BIG IDEA

Some functions mutate the list and don't return anything.

We use these functions for their side effect.

OPERATION ON LISTS: append

```
L = [2,1,3]
L.append(5)
```

function arguments

a function that works on an object of this type

an object of some type

- What is the dot?
  - Lists are Python objects, everything in Python is an object
  - Objects have **data**
  - Object types also have **associated operations**
  - Access this information by `object_name.do_something()`
  - Equivalent to calling `append` with arguments `L` and `5`

YOU TRY IT!

- Write a function that meets these specs:

```
def make_ordered_list(n):
    """ n is a positive int
    Returns a list containing all ints in order
    from 0 to n (inclusive)
    """
```

YOU TRY IT!

- Write a function that meets the specification.

```
def remove_elem(L, e):
    """
    L is a list
    Returns a new list with elements in the same order as L
    but without any elements equal to e.
    """


L = [1,2,2,2]
print(remove_elem(L, 2))    # prints [1]
```

STRINGS to LISTS

- Convert **string to list** with `list(s)`
  - Every character from `s` is an element in a list
- Use `s.split()`, to **split a string on a character** parameter, splits on spaces if called without a parameter

```
s = "I<3 cs &u?"                 → s is a string
L = list(s)            → L is ['I','<','3',' ','c','s',' ','&','u','?']

L1 = s.split(' ')         → L1 is ['I<3','cs','&u?']
L2 = s.split('<')         → L2 is ['I', '3 cs &u?']
```

LISTS to STRINGS

- Convert a **list of strings back to string**
- Use `''.join(L)` to turn a **list of strings into a bigger string**
- Can give a character in quotes to add char between every element

```
L = ['a','b','c']               → L is a list
A = ''.join(L)                  → A is "abc"
B = '_'.join(L)                 → B is "a_b_c"
C = ''.join([1,2,3])            → an error
C = ''.join(['1','2','3']       → C is "123" a string!
```

YOU TRY IT!

- Write a function that meets these specs:

```
def count_words(sen):
    """ sen is a string representing a sentence
    Returns how many words are in s (i.e. a word is a
    a sequence of characters between spaces. """

print(count_words("Hello it's me"))
```

A FEW INTERESTING LIST OPERATIONS

- **Add** an element to end of list with `L.append(element)`
  - **mutates** the list
- `sort()`
  - `L = [4,2,7]`
    `L.sort()`
  - **Mutates** L
- `reverse()`
  - `L = [4,2,7]`
    `L.reverse()`
  - **Mutates** L
- `sorted()`
  - `L = [4,2,7]`
  - `L_new = sorted(L)`
  - Returns a sorted version of L (**no mutation**!)

Remember . notation: object.operation()
Do `append` operation on `L`, with parameter `element`

MUTABILITY

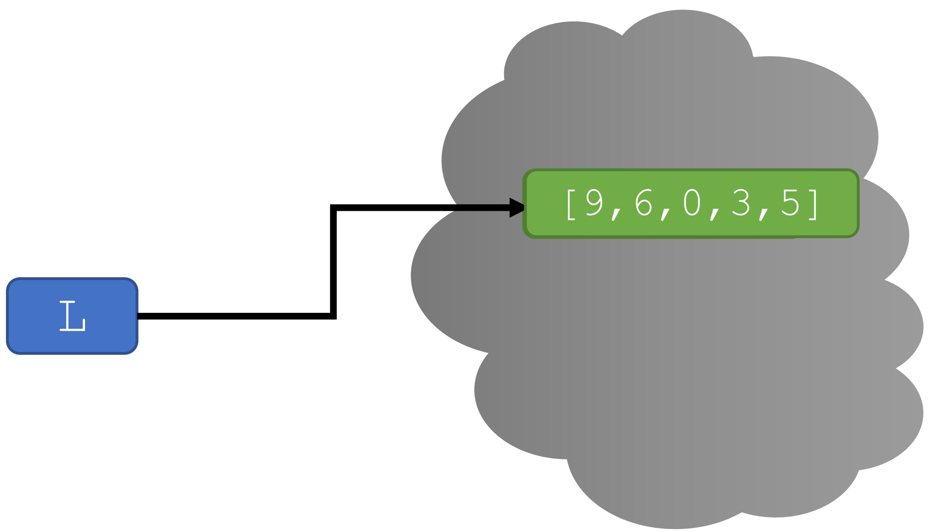
`append`, `sort`, `reverse`
all used for side effect

```
L=[9,6,0,3]
L.append(5)
```

`a = sorted(L)` → returns a **new** sorted list, does **not mutate** `L`

`b = L.sort()` → **mutates** `L` to be `[0,3,5,6,9]` and returns None

`L.reverse()` → **mutates** `L` to be `[9,6,5,3,0]` and returns None



MUTABILITY

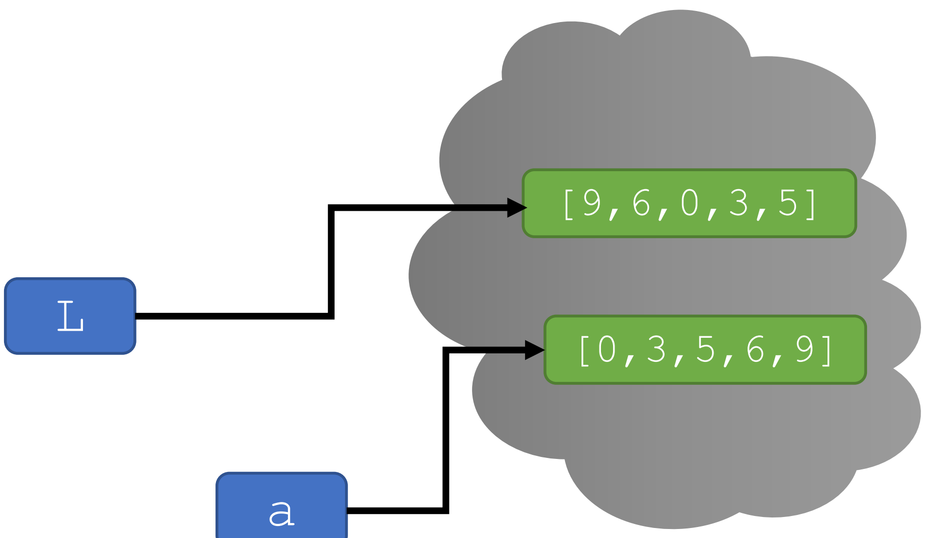
`append`, `sort`, `reverse`
all used for side effect

```
L=[9,6,0,3]
L.append(5)
```

$\boxed{\texttt{a = sorted(L)}}$ → returns a **new** sorted list, does **not mutate** `L`

`b = L.sort()` → **mutates** `L` to be `[0,3,5,6,9]` and returns None

`L.reverse()` → **mutates** `L` to be `[9,6,5,3,0]` and returns None



MUTABILITY

`append, sort, reverse`
all used for side effect

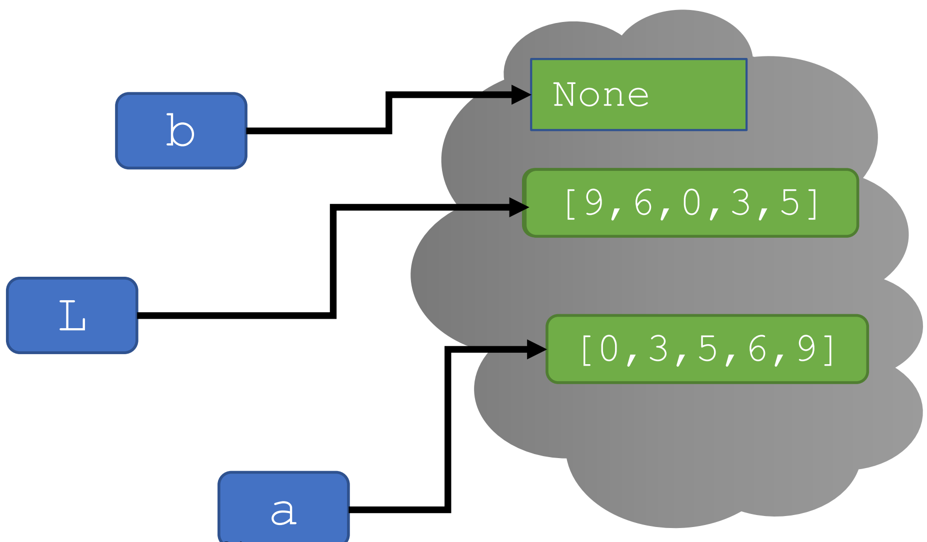
```
L=[9,6,0,3]
L.append(5)
```

`a = sorted(L)` → returns a **new** sorted list, does **not mutate** `L`

$\boxed{\texttt{b = } \boxed{\texttt{L.sort()}}}$ → **mutates** `L` to be `[0,3,5,6,9]` and returns None

`L.reverse()` → **mutates** `L` to be `[9,6,5,3,0]` and returns None

Never do this.
Just use L.sort()!



MUTABILITY

append, sort, reverse
all used for **side effect**

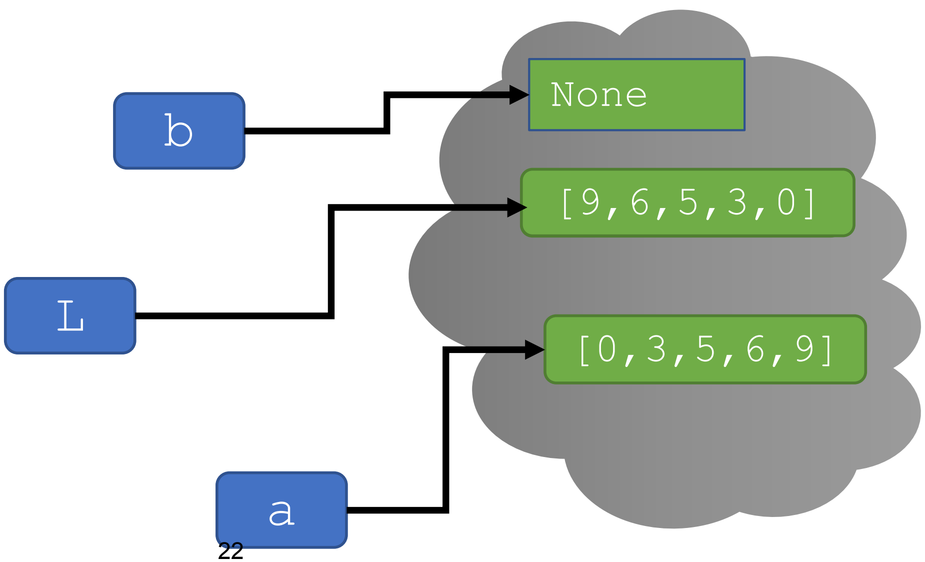
```
L=[9,6,0,3]
L.append(5)
```

`a = sorted(L)` → returns a **new** sorted list, does **not mutate** `L`

`b = L.sort()` → **mutates** `L` to be `[0,3,5,6,9]` and returns None

`L.reverse()` → **mutates** `L` to be `[9,6,5,3,0]` and returns None

Remember, we have to invoke the function even if it takes no arguments



YOU TRY IT!

- Write a function that meets these specs:

```
def sort_words(sen):
    """ sen is a string representing a sentence
    Returns a list containing all the words in sen but
    sorted in alphabetical order. """

print(sort_words("look at this photograph"))
```

BIG IDEA

Functions with side effects mutate inputs.

You can write your own!

LISTS SUPPORT ITERATION

- Let’s write a **function that mutates the input**
- Example: square every element of a list, mutating original list

```
def square_list(L):
    for elem in L:
        # ?? How to do L[index] = the square ??
        # ?? elem is an element in L, not the index :(
```

- Solutions (we’ll go over option 2, try the others on your own!):
  - Option 1: Make a **new variable** representing the index, initialized to 0 before the loop and incremented by 1 in the loop.
  - Option 2: **Loop over the index** not the element, and use L[index] to get the element
  - Option 3: Use **enumerate** in the for loop (I leave this option to you to look up). i.e. `for i,e in enumerate(L)`

LISTS SUPPORT ITERATION

- Example: square every element of a list, mutating original list

```
def square_list(L):
    for i in range(len(L)):
        L[i] = L[i]**2
```

To change elements of list, need to loop over indices into list

An assignment statement. L[i] is not a name, but points to a particular spot in the list data structure.

L[i] is the element

- Note, **no return**!

TRACE the CODE with an EXAMPLE

- Example: square every element of a list, mutating original list

```
def square_list(L):
    for i in range(len(L)):
        L[i] = L[i]**2
```

Suppose L is [2,3,4]

i is 0: L is mutated to [4, 3, 4]

i is 1: L is mutated to [4, 9, 4]

i is 2: L is mutated to [4, 9, 16]

TRACE the CODE with an EXAMPLE

- Example: square every element of a list, mutating original list

```
def square_list(L):
    for i in range(len(L)):
        L[i] = L[i]**2
```

The function mutates the input object passed in (Lin)

```
Lin = [2,3,4]
print("before fcn call:",Lin)   # prints [2,3,4]
square_list(Lin)
print("after fcn call:",Lin)    # prints [4,9,16]
```

No variable to assign function call to!

BIG IDEA

Functions that mutate the input likely.....

Iterate over len(L) not L.

Return None, so the function call does not need to be saved.

MUTATION

- Lists are **mutable** structures
- There are many advantages to being able to **change a portion** of a list
  - Suppose I have a very long list (e.g. of personnel records) and I want to update one element. Without mutation, I would have to copy the entire list, with a new version of that record in the right spot. A mutable structure lets me change just that element
- But, this ability can also introduce unexpected challenges

TRICKY EXAMPLES OVERVIEW

- TRICKY EXAMPLE 1:
  - A loop iterates over **indices of L** and **mutates L** each time (adds more elements).
- TRICKY EXAMPLE 2:
  - A loop iterates over **L's elements** directly and **mutates L** each time (adds more elements).
- TRICKY EXAMPLE 3:
  - A loop iterates over **L's elements** directly but **reassigns L** to a new object each time
- TRICKY EXAMPLE 4 (next time):
  - A loop iterates over **L's elements** directly and mutates L by **removing elements**.

TRICKY EXAMPLE 1: append

- **Range returns something that behaves like a tuple** (but isn't – it returns an *iterable*)
  - Returns the first element, and an iteration method by which subsequent elements are generated as needed

`range(4)` → *kind of like* tuple `(`$\boxed{\texttt{0}}$`,`$\boxed{\texttt{1}}$`,`$\boxed{\texttt{2}}$`,`$\boxed{\texttt{3}}$`)`

`range(2,9,2)` → *kind of like* tuple `(2,4,6,8)`

```
L = [1,2,3,4]
for i in range(len(L)):
    L.append(i)
print(L)
```

Iteration sequence is pre-determined at beginning of loop

1st time: L is [1, 2, 3, 4, 0]

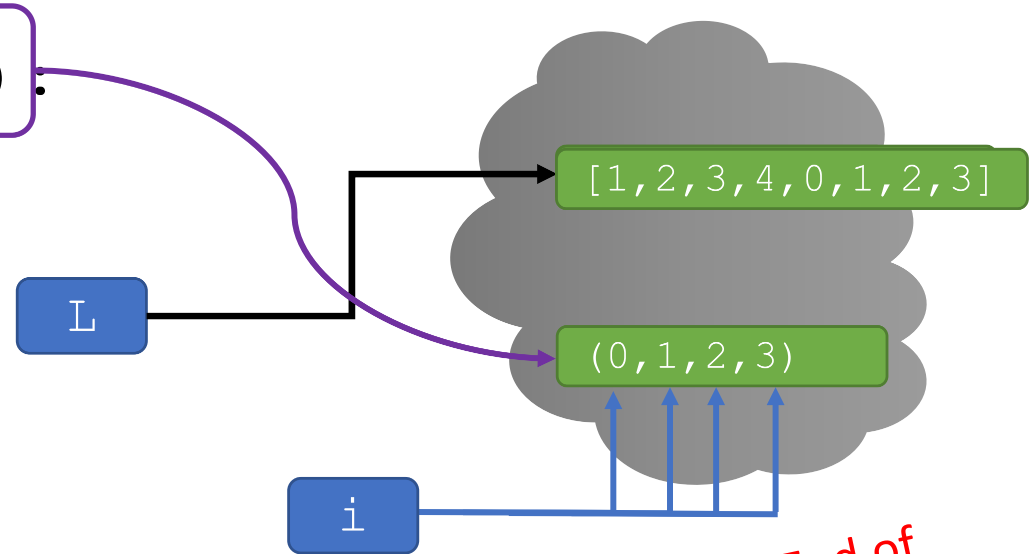
2nd time: L is [1, 2, 3, 4, 0, 1]

3rd time: L is [1, 2, 3, 4, 0, 1, 2]

4th time: L is [1, 2, 3, 4, 0, 1, 2, 3]

TRICKY EXAMPLE 1: append

```
L = [1,2,3,4]
for i in range(len(L)):
    L.append(i)
    print(L)
```



1st time: L is [1, 2, 3, 4, 0]

2nd time: L is [1, 2, 3, 4, 0, 1]

3rd time: L is [1, 2, 3, 4, 0, 1, 2]

4th time: L is [1, 2, 3, 4, 0, 1, 2, 3]

End of iteration

`i` iterates over a “tuple” created by range; mutation of `L` does not affect this “tuple”

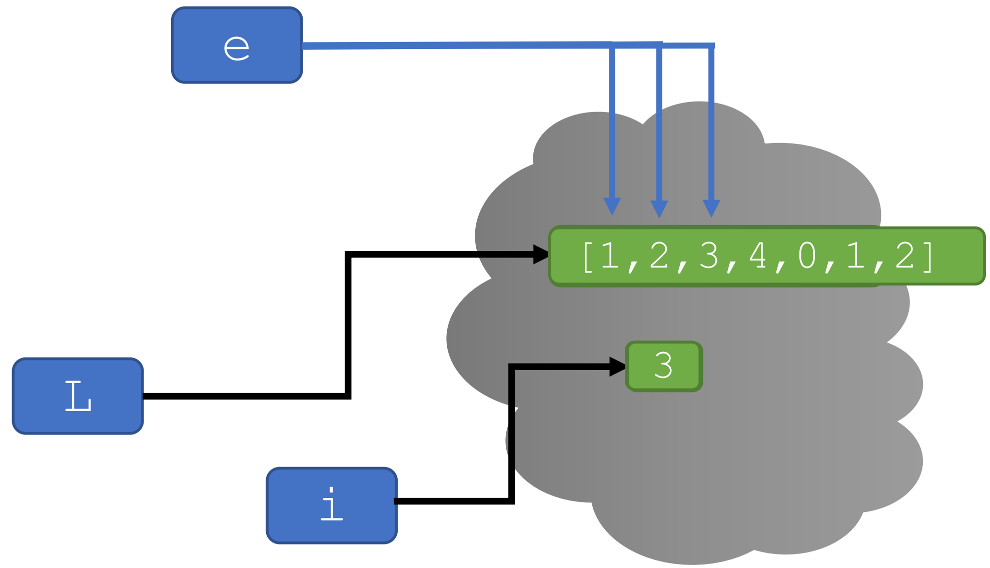
TRICKY EXAMPLE 2: append

Looks similar **but** ...

```
L = [1,2,3,4]
i = 0
for e in L:
    L.append(i)
    i += 1
print(L)
```

Originally [1,2,3,4]

L is **mutated** each iteration



1st time: L is [1, 2, 3, 4, 0]

2nd time: L is [1, 2, 3, 4, 0, 1]

3rd time: L is [1, 2, 3, 4, 0, 1, 2]

4th time: L is [1, 2, 3, 4, 0, 1, 2, 3]

NEVER STOPS!

In previous example, L was accessed at onset to create a range iterable; in this example, the loop is directly accessing indices into L

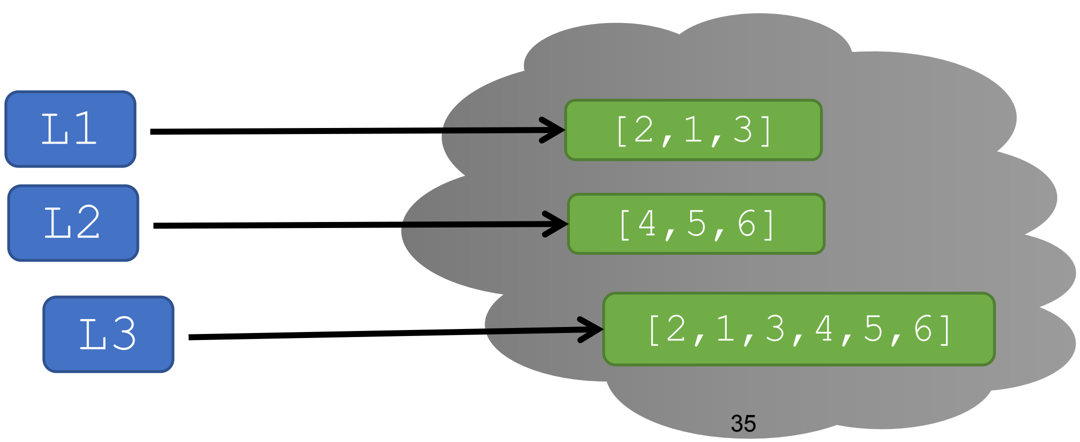
COMBINING LISTS

Remember strings

- **Concatenation**, + operator, creates a **new** list, with copies
- **Mutate** list with `L.extend(some_list)` (copy of `some_list`)

```
L1 = [2,1,3]
L2 = [4,5,6]
L3 = L1 + L2
```

→ `L3` is `[2,1,3,4,5,6]`



concatenation creates **new** list with copies

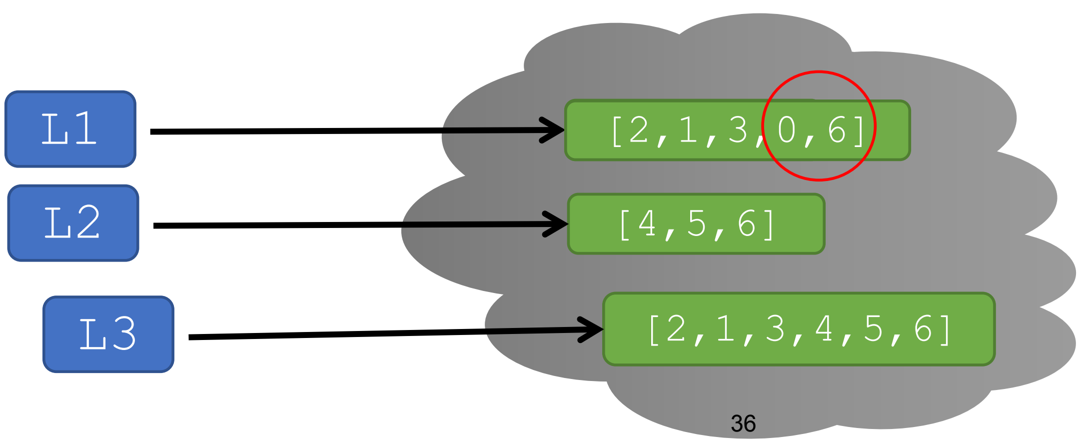
COMBINING LISTS

- **Concatenation**, + operator, creates a **new** list, with copies
- **Mutate** list with `L.extend(some_list)` (copy of `some_list`)

```
L1 = [2,1,3]
L2 = [4,5,6]
L3 = L1 + L2
L1.extend([0,6])
```

→ `L3` is `[2,1,3,4,5,6]`

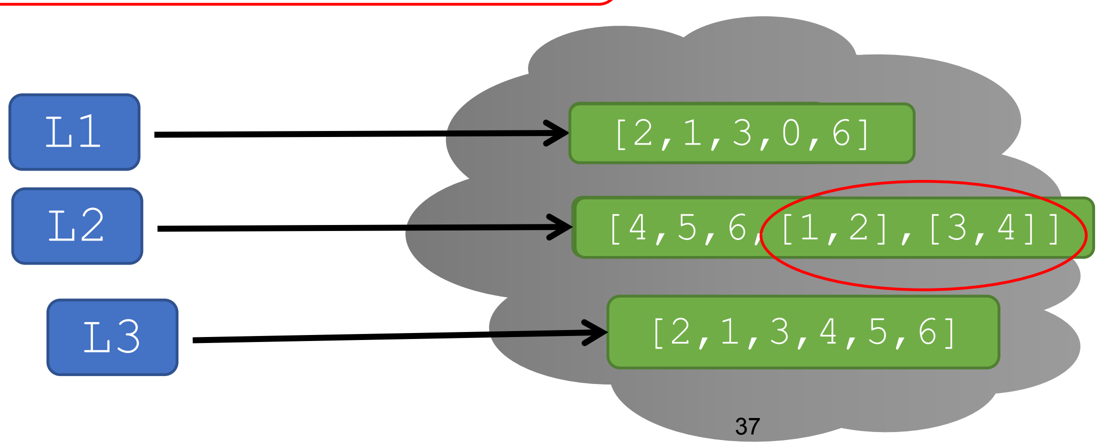
→ mutate `L1` to `[2,1,3,0,6]`



COMBINING LISTS

- **Concatenation**, + operator, creates a **new** list, with copies
- **Mutate** list with `L.extend(some_list)` (copy of `some_list`)

```
L1 = [2,1,3]
L2 = [4,5,6]
L3 = L1 + L2                 → L3 is [2,1,3,4,5,6]
L1.extend([0,6])             → mutate L1 to [2,1,3,0,6]
L2.extend([[1,2],[3,4]])     → mutates L2 to [4,5,6,[1,2],[3,4]]
```



Extending by a list of lists gives us new list elements

TRICKY EXAMPLE 3: combining

```
L = [1,2,3,4]
for e in L:
    L = L + L
    print(L)
```

Originally [1,2,3,4]

1st time: **new** L is [1, 2, 3, 4, 1, 2, 3, 4]

2nd time: **new** L is [1, 2, 3, 4, 1, 2, 3, 4,
1, 2, 3, 4, 1, 2, 3, 4]

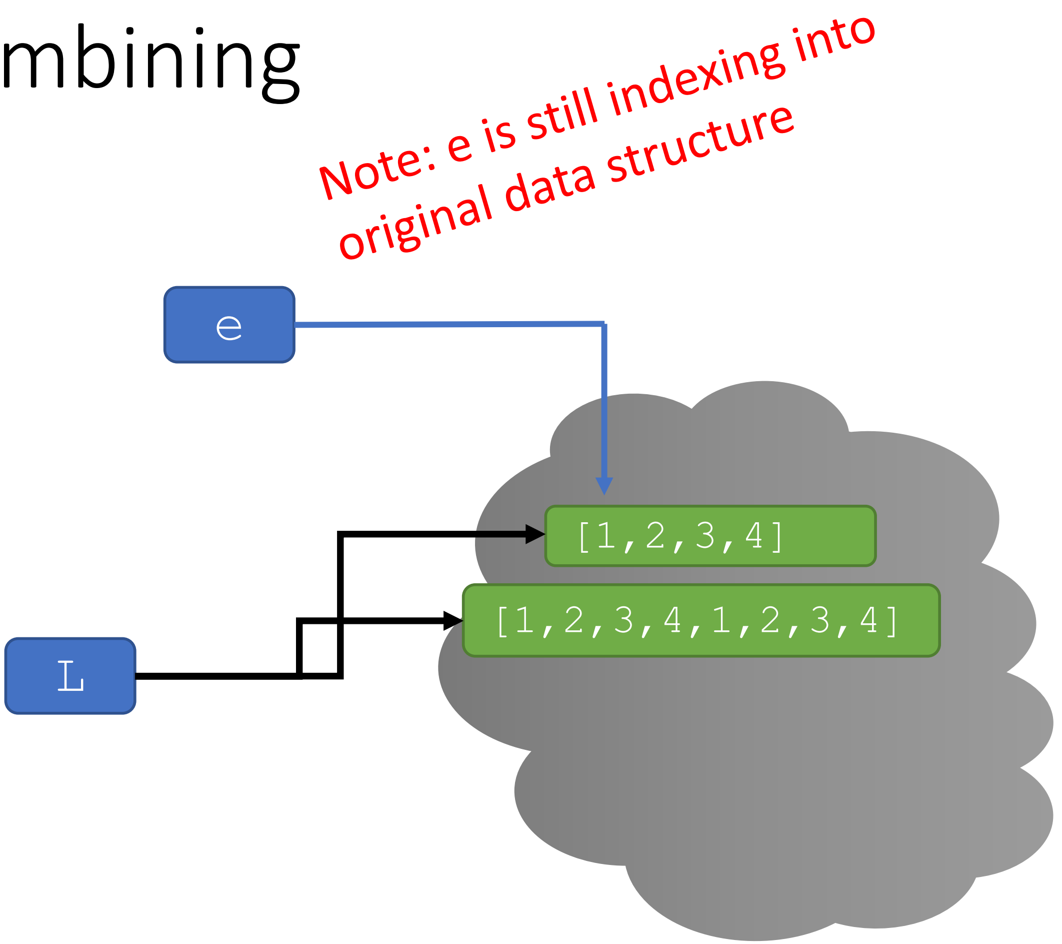
3rd time: **new** L is [1, 2, 3, 4, 1, 2, 3, 4,
1, 2, 3, 4, 1, 2, 3, 4
1, 2, 3, 4, 1, 2, 3, 4,
1, 2, 3, 4, 1, 2, 3, 4]

4th time: **new** L is [1, 2, 3, 4, 1, 2, 3, 4,
1, 2, 3, 4, 1, 2, 3, 4
1, 2, 3, 4, 1, 2, 3, 4,
1, 2, 3, 4, 1, 2, 3, 4
1, 2, 3, 4, 1, 2, 3, 4,
1, 2, 3, 4, 1, 2, 3, 4
1, 2, 3, 4, 1, 2, 3, 4,
1, 2, 3, 4, 1, 2, 3, 4]

L is **bound to a new object** each iteration; but looping of e walks down structure pointed to when called, so iterates only 4 times, over original [1,2,3,4]

TRICKY EXAMPLE 3: combining

```
L = [1,2,3,4]
for e in L:
    L = L + L
    print(L)
```

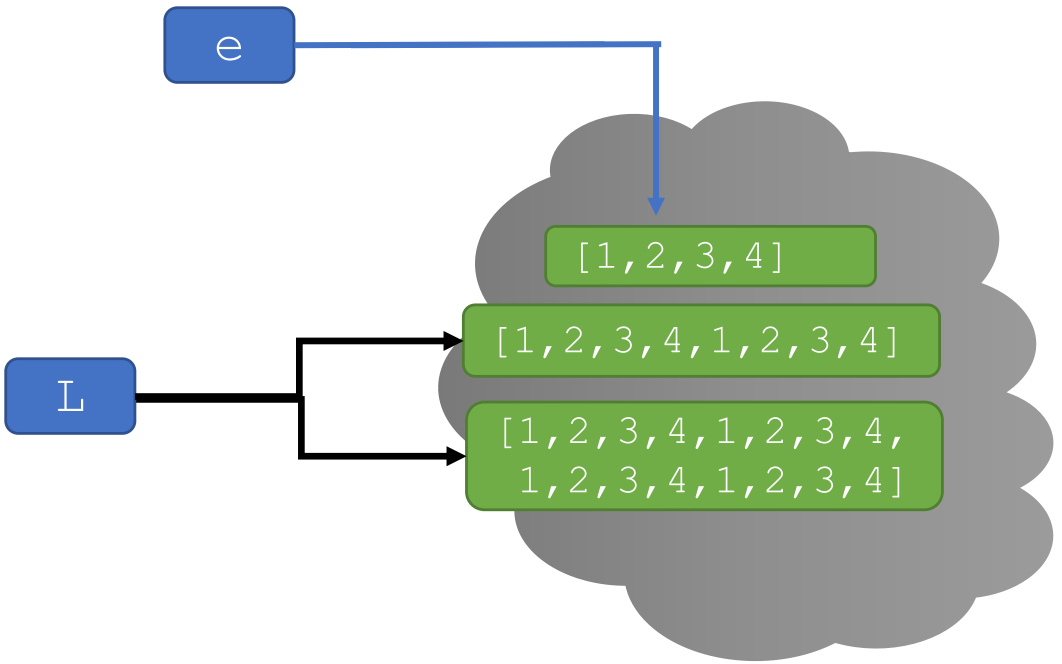


1[st] time: **new** L is [1, 2, 3, 4, 1, 2, 3, 4]

TRICKY EXAMPLE 3: combining

Note: e is still indexing into original data structure

```
L = [1,2,3,4]
for e in L:
    L = L + L
    print(L)
```



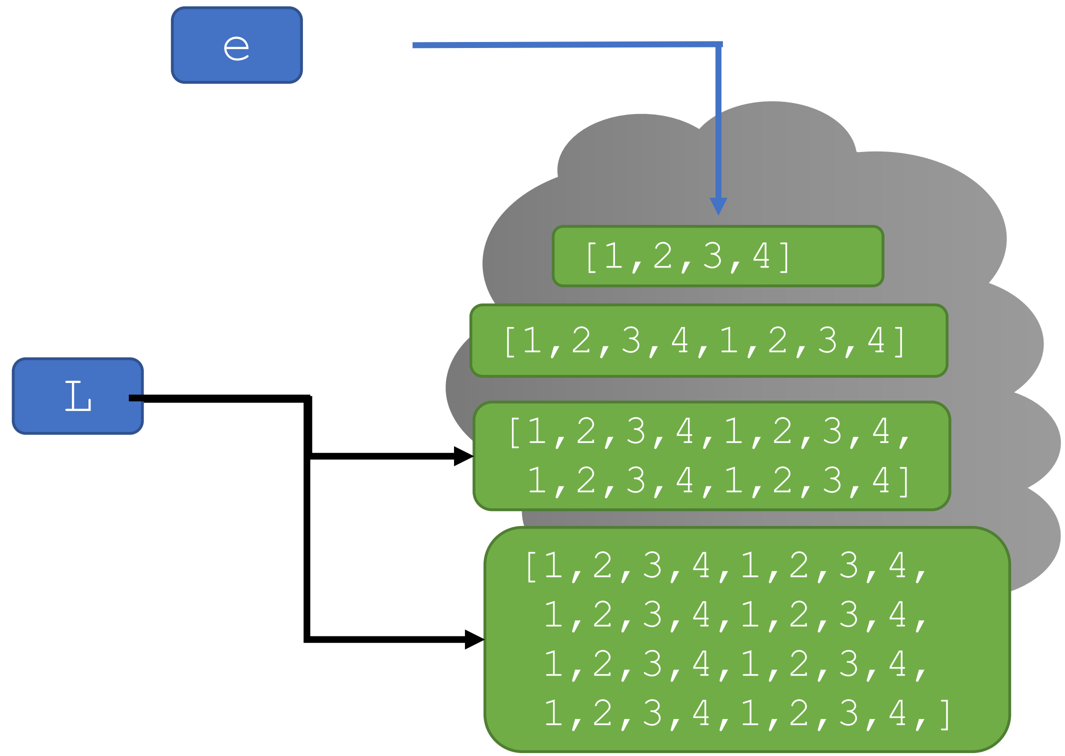
1st time: **new** L is [1, 2, 3, 4, 1, 2, 3, 4]

2nd time: **new** L is [1, 2, 3, 4, 1, 2, 3, 4,
1, 2, 3, 4, 1, 2, 3, 4]

TRICKY EXAMPLE 3: combining

Note: e is still indexing into original data structure

```
L = [1,2,3,4]
for e in L:
    L = L + L
    print(L)
```



1st time: **new** L is [1, 2, 3, 4, 1, 2, 3, 4]

2nd time: **new** L is [1, 2, 3, 4, 1, 2, 3, 4,
1, 2, 3, 4, 1, 2, 3, 4]

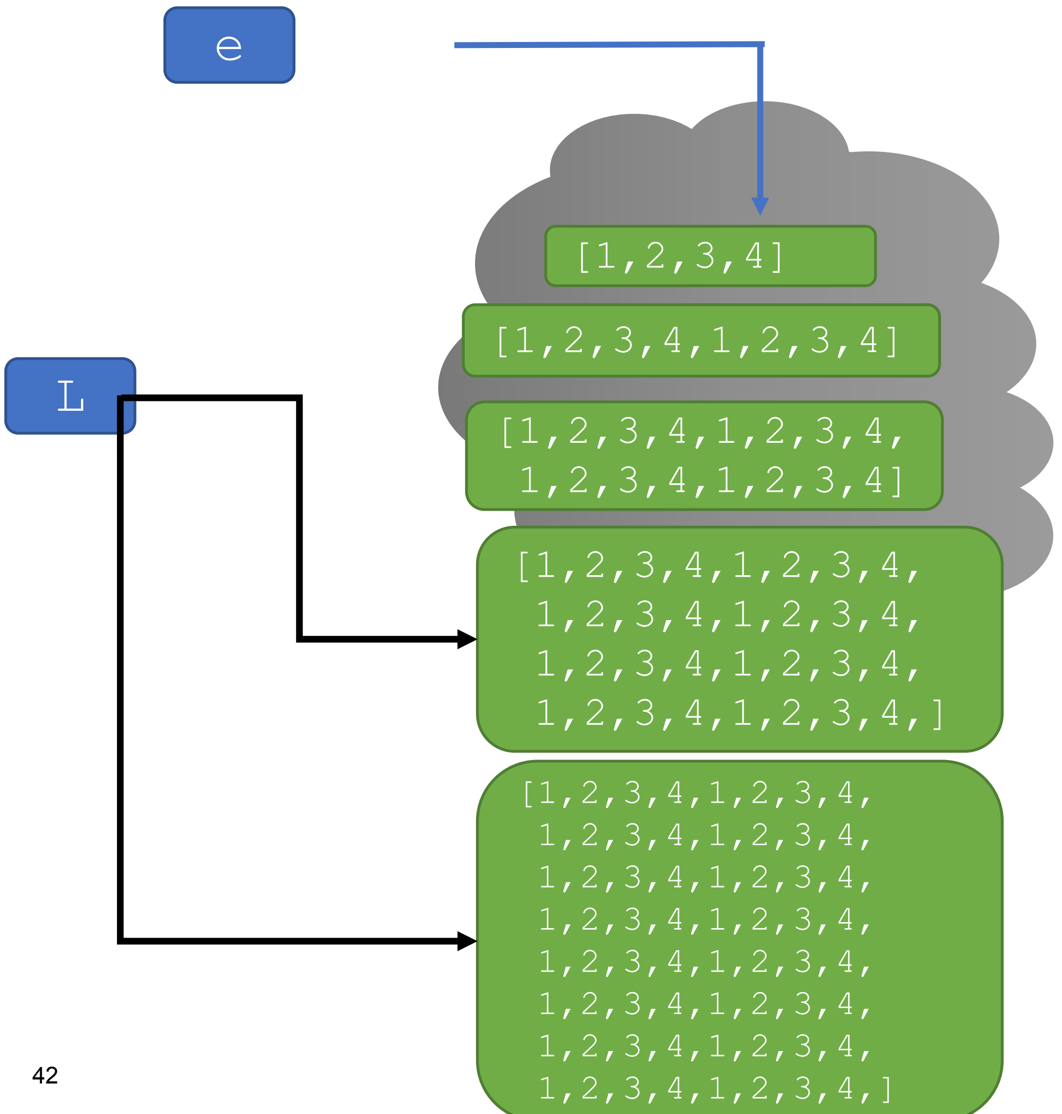
3rd time: **new** L is [1, 2, 3, 4, 1, 2, 3, 4,
1, 2, 3, 4, 1, 2, 3, 4 ,
1, 2, 3, 4, 1, 2, 3, 4,
1, 2, 3, 4, 1, 2, 3, 4]

TRICKY EXAMPLE 3: combining

```
L = [1,2,3,4]
for e in L:
    L = L + L
    print(L)
```

4th time: **new** L is [1, 2, 3, 4, 1, 2, 3, 4,
1, 2, 3, 4, 1, 2, 3, 4 ,
1, 2, 3, 4, 1, 2, 3, 4,
1, 2, 3, 4, 1, 2, 3, 4
1, 2, 3, 4, 1, 2, 3, 4,
1, 2, 3, 4, 1, 2, 3, 4 ,
1, 2, 3, 4, 1, 2, 3, 4,
1, 2, 3, 4, 1, 2, 3, 4]

Note: e is still indexing into original data structure



EMPTY OUT A LIST AND CHECKING THAT IT'S THE SAME OBJECT

- You can **mutate a list to remove all its elements**
  - This **does not make a new empty list**!
- Use `L.clear()`
- How to check that it's **the same object in memory**?
  - Use the id() function
  - Try this in the console

```
>>> L = [4,5,6]
>>> id(L)
>>> L.append(8)
>>> id(L)
>>> L.clear()
>>> id(L)
```

Same!

```
>>> L = [4,5,6]
>>> id(L)
>>> L.append(8)
>>> id(L)
>>> L = []
>>> id(L)
```

Different!

SUMMARY

- Lists and tuples provide a way to organize data that naturally supports iterative functions
- Tuples are **immutable** (like strings)
  - Tuples are useful when you have **data that doesn't need to change**. e.g. (latitude, longitude) or (page #, line #)
- Lists are **mutable**
  - You can modify the object by **changing an element** at an index
  - You can modify the object by **adding elements** to the end
  - Will see many more operations on lists next time
  - Lists are useful in **dynamic situations**. e.g. a list of daily top 40 songs or a list of recently watched movies

ALIASING, CLONING

(download slides and .py files to follow along)

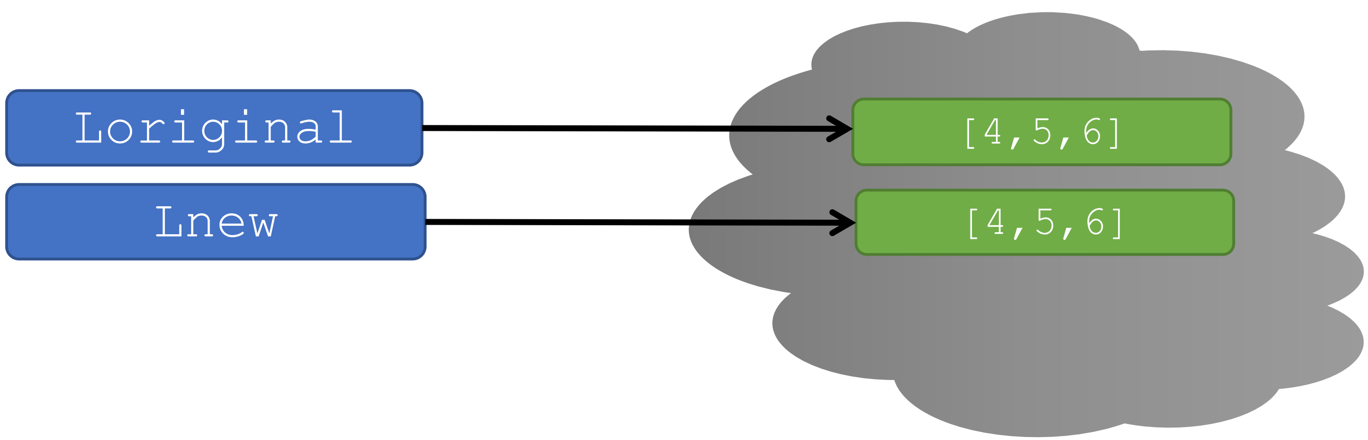
6.100L Lecture 11

Ana Bell

MAKING A COPY OF THE LIST

- Can **make a copy of a list object** by duplicating all elements (top-level) into a new list object
- `Lcopy = L[:]`
  - Equivalent to looping over L and appending each element to Lcopy
  - This does not make a copy of elements that are lists (will see how to do this at the end of this lecture)

```
Loriginal = [4,5,6]
Lnew = Loriginal[:]
```



YOU TRY IT!

- Write a function that meets the specification.
- Hint. Make a copy to save the elements. The use L.clear() to empty out the list and repopulate it with the ones you're keeping.

```
def remove_all(L, e):
    """
    L is a list
    Mutates L to remove all elements in L that are equal to e
    Returns None
    """


L = [1,2,2,2]
remove_all(L, 2)
print(L)     # prints [1]
```

OPERATION ON LISTS: `remove`

- Delete element at a **specific index** with `del(L[index])`
- Remove element at **end of list** with `L.pop()`, returns the removed element (can also call with specific index: `L.pop(3)`)
- Remove a **specific element** with `L.remove(element)`
  - Looks for the element and removes it (mutating the list)
  - If element occurs multiple times, removes first occurrence
  - If element not in list, gives an error

```
L = [2,1,3,6,3,7,0] # do below in order
L.remove(2)    → mutates L = [1,3,6,3,7,0]
L.remove(3)    → mutates L = [1,6,3,7,0]
del(L[1])      → mutates L = [1,3,7,0]
a = L.pop()    → returns 0 and mutates L = [1,3,7]
```

all these operations **mutate** the list

EXERCISE WITH REMOVE INSTEAD OF COPY AND CLEAR

- Rewrite the code to remove `e` as long as we still had it in the list
- It works well!

```
def remove_all(L, e):
    """
    L is a list
    Mutates L to remove all elements in L that are equal to e
    Returns None.
    """
    while e in L:
        L.remove(e)
```

EXERCISE WITH REMOVE INSTEAD OF COPY AND CLEAR

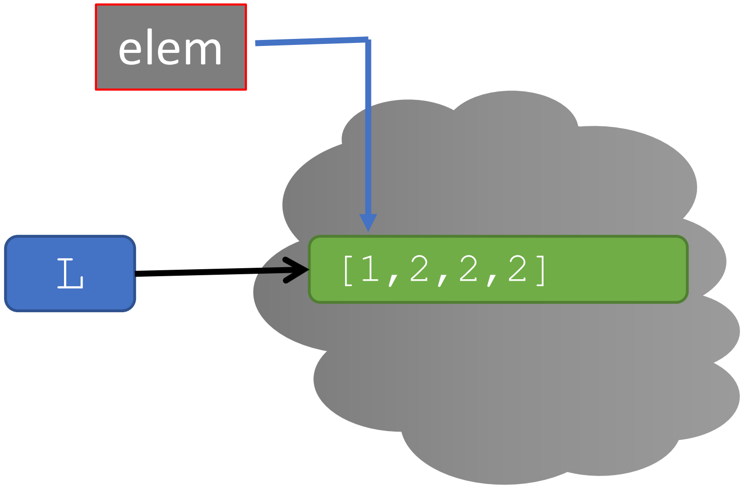
- What if the code was this:

```
def remove_all(L, e):
    """
    L is a list
    Mutates L to remove all elements in L that are equal to e
    Returns None.
    """
    for elem in L:
        if elem == e:
            L.remove(e)

L = [1,2,2,2]
remove_all(L, 2)
print(L)    # should print [1]
```

Actually prints [1,2]

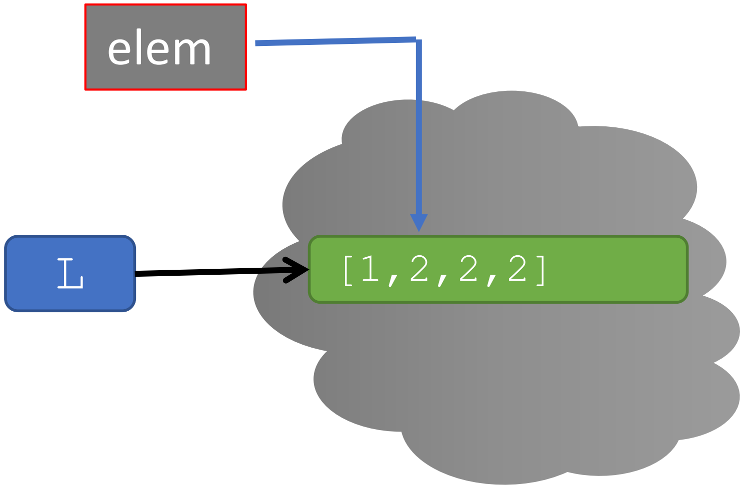
EXERCISE WITH REMOVE INSTEAD OF COPY AND CLEAR

```
def remove_all(L, e):
    """
    L is a list
    Mutates L to remove all elements in L that are equal to e
    Returns None.
    """
    for elem in L:
        if elem == e:
            L.remove(e)

L = [1,2,2,2]
remove_all(L, 2)
print(L)    # should print [1]
```



EXERCISE WITH REMOVE INSTEAD OF COPY AND CLEAR

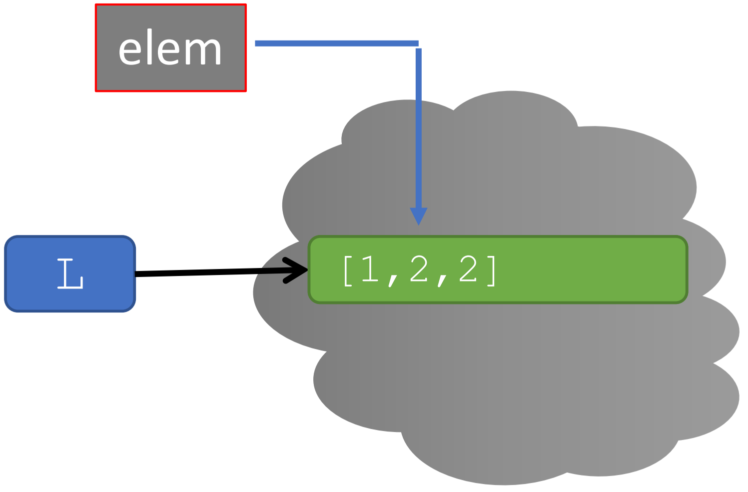
```
def remove_all(L, e):
    """
    L is a list
    Mutates L to remove all elements in L that are equal to e
    Returns None.
    """
    for elem in L:
        if elem == e:
            L.remove(e)

L = [1,2,2,2]
remove_all(L, 2)
print(L)     # should print [1]
```



EXERCISE WITH REMOVE INSTEAD OF COPY AND CLEAR

```
def remove_all(L, e):
    """
    L is a list
    Mutates L to remove all elements in L that are equal to e
    Returns None.
    """
    for elem in L:
        if elem == e:
            L.remove(e)

L = [1,2,2,2]
remove_all(L, 2)
print(L)     # should print [1]
```

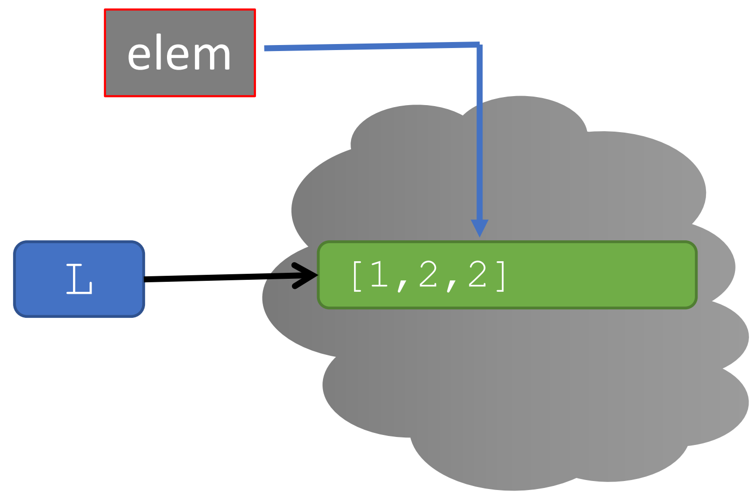
Removes the 2, but doesn't shift the pointer back!



EXERCISE WITH REMOVE INSTEAD OF COPY AND CLEAR

```
def remove_all(L, e):
    """
    L is a list
    Mutates L to remove all elements in L that are equal to e
    Returns None.
    """
    for elem in L:
        if elem == e:
            L.remove(e)

L = [1,2,2,2]
remove_all(L, 2)
print(L)     # should print [1]
```

elem moves forward in the sequence

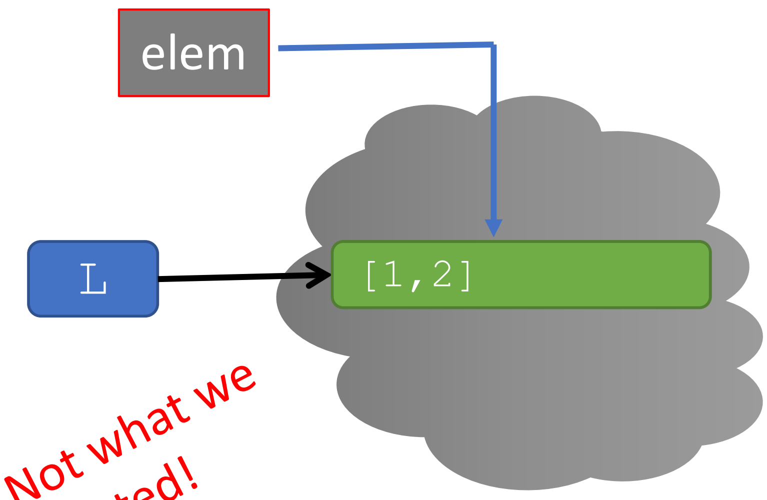


EXERCISE WITH REMOVE INSTEAD OF COPY AND CLEAR

- It's not correct! We **removed items as we iterated over the list**!

```
def remove_all(L, e):
    """
    L is a list
    Mutates L to remove all elements in L that are equal to e
    Returns None.
    """
    for elem in L:
        if elem == e:
            L.remove(e)

L = [1,2,2,2]
remove_all(L, 2)
print(L)    # should print [1]
```

Remove the 2, and done



Not what we wanted!

TRICKY EXAMPLES OVERVIEW

- TRICKY EXAMPLE 1:
  - A loop iterates over **indices of L** and **mutates L** each time (adds more elements).
- TRICKY EXAMPLE 2:
  - A loop iterates over **L's elements** directly and **mutates L** each time (adds more elements).
- TRICKY EXAMPLE 3:
  - A loop iterates over **L's elements** directly but **reassigns L** to a new object each time
- TRICKY EXAMPLE 4:
  - A loop iterates over **L's elements** directly and mutates L by **removing elements**.

TRICKY EXAMPLE 4

PYTHON TUTOR LINK to see step-by-step

- Want to mutate L1 to remove any elements that are also in L2

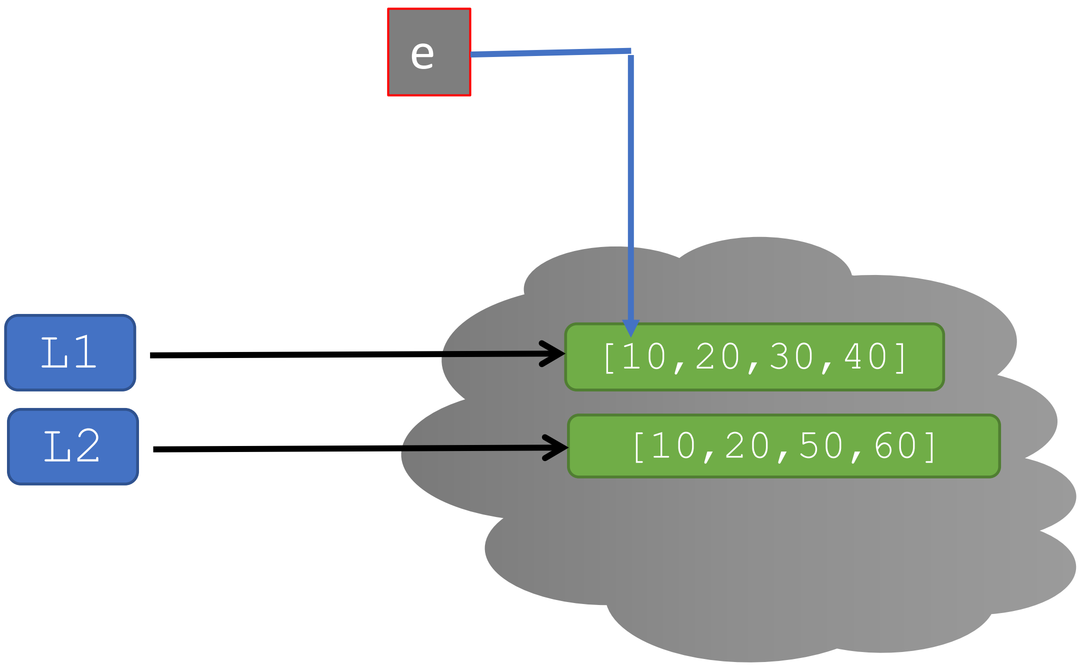
```
def remove_dups(L1, L2):
    for e in L1:
        if e in L2:
            L1.remove(e)

L1 = [10, 20, 30, 40]
L2 = [10, 20, 50, 60]
remove_dups(L1, L2)
```

Want a function that returns a list, where every element also in a second list is removed.

- `L1` is `[20,30,40]` not `[30,40]` Why?
  - You are **mutating a list as you are iterating over it**
  - Python uses an internal counter. Tracks of index in the loop over list L1
  - Mutating changes the list but Python doesn't update the counter
  - Loop never sees element 20

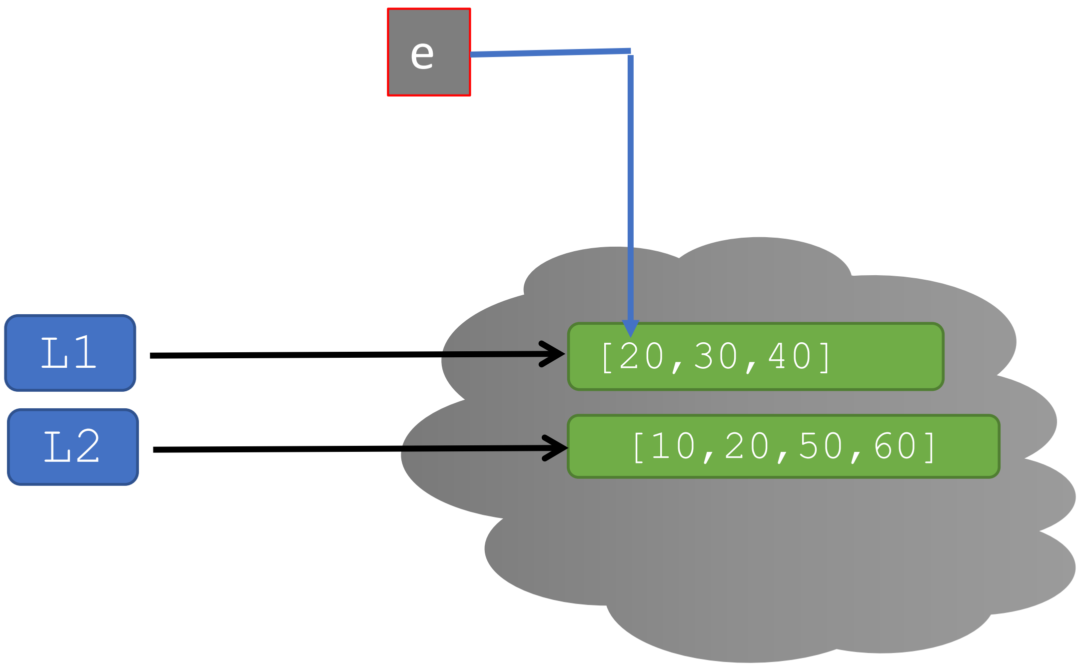
MUTATION AND ITERATION WITHOUT CLONE

```
def remove_dups(L1, L2):
    for e in L1:
        if e in L2:
            L1.remove(e)

L1 = [10, 20, 30, 40]
L2 = [10, 20, 50, 60]
remove_dups(L1, L2)
```



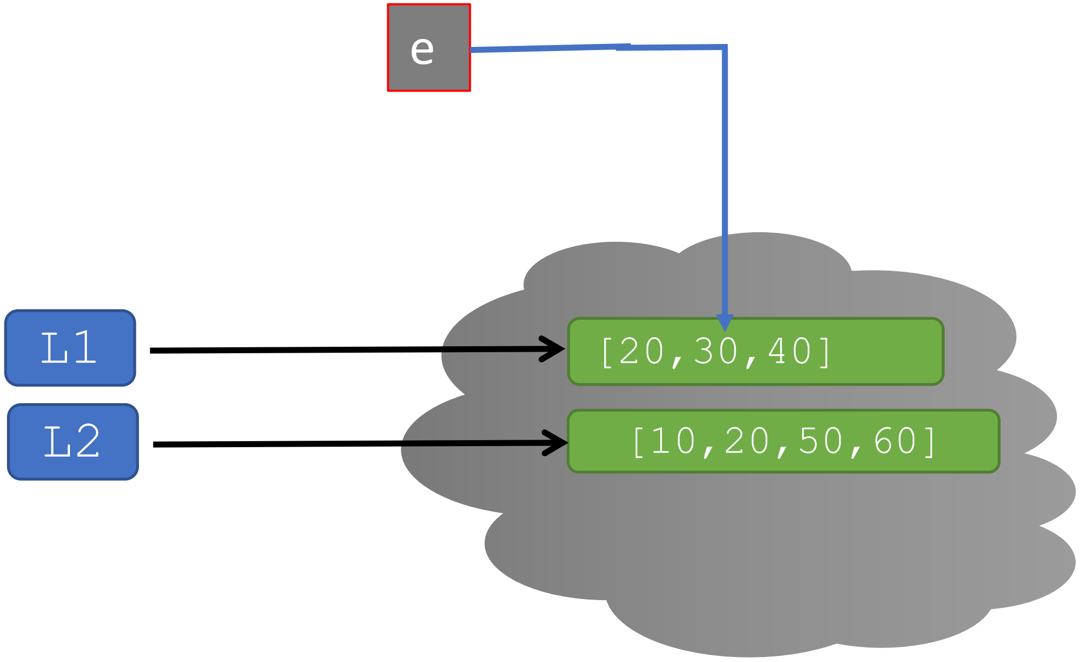
MUTATION AND ITERATION WITHOUT CLONE

```
def remove_dups(L1, L2):
    for e in L1:
        if e in L2:
            L1.remove(e)

L1 = [10, 20, 30, 40]
L2 = [10, 20, 50, 60]
remove_dups(L1, L2)
```



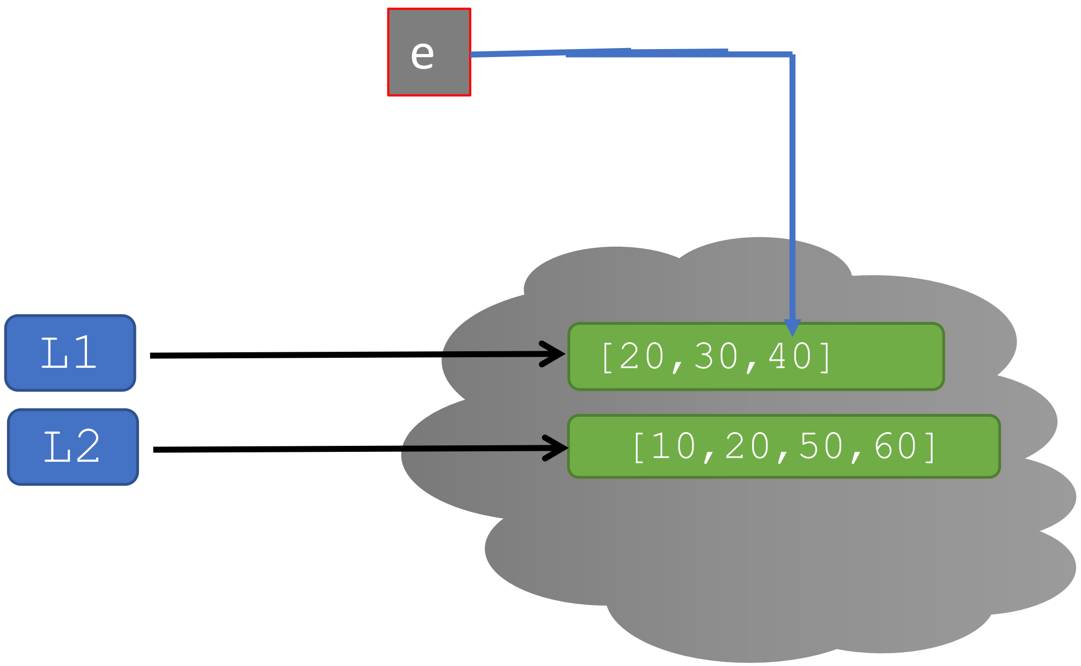
MUTATION AND ITERATION WITHOUT CLONE

```
def remove_dups(L1, L2):
    for e in L1:
        if e in L2:
            L1.remove(e)

L1 = [10, 20, 30, 40]
L2 = [10, 20, 50, 60]
remove_dups(L1, L2)
```



MUTATION AND ITERATION WITHOUT CLONE

```
def remove_dups(L1, L2):
    for e in L1:
        if e in L2:
            L1.remove(e)

L1 = [10, 20, 30, 40]
L2 = [10, 20, 50, 60]
remove_dups(L1, L2)
```



MUTATION AND ITERATION **WITH CLONE**

`L1_copy = L1[:]`

- Make a **clone** with [:]

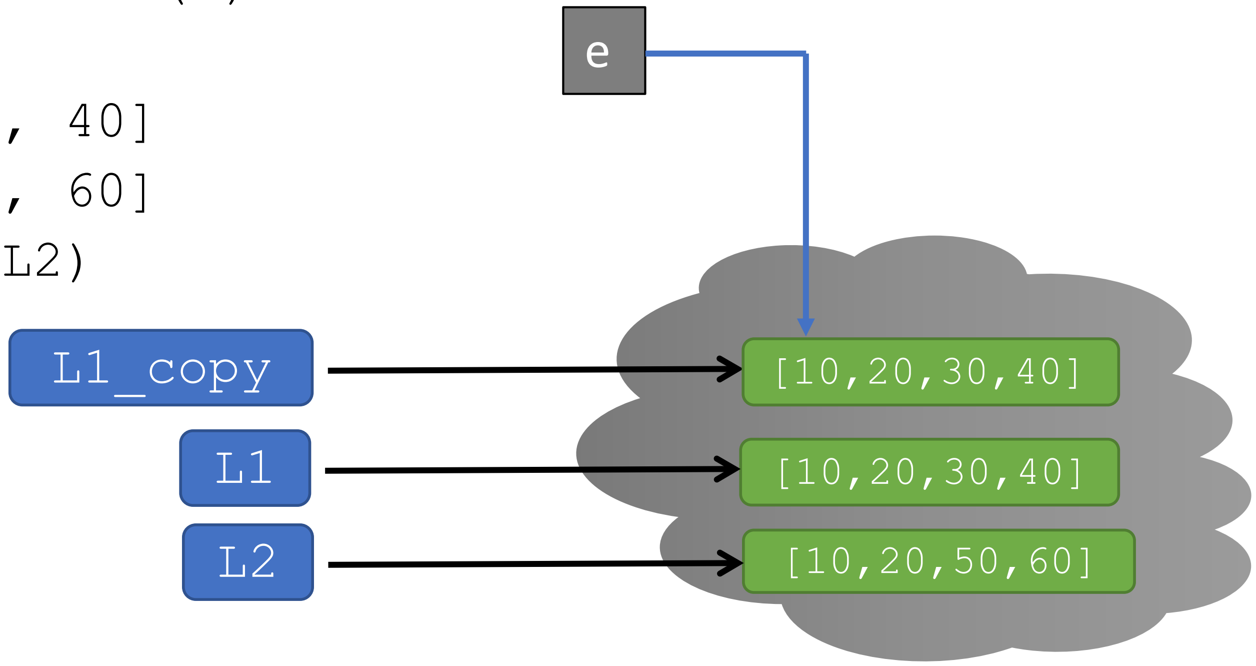
```
def remove_dups(L1, L2):
    for e in L1:
        if e in L2:
            L1.remove(e)
```

```
def remove_dups(L1, L2):
    L1_copy = L1[:]
    for e in L1_copy:
        if e in L2:
            L1.remove(e)
```

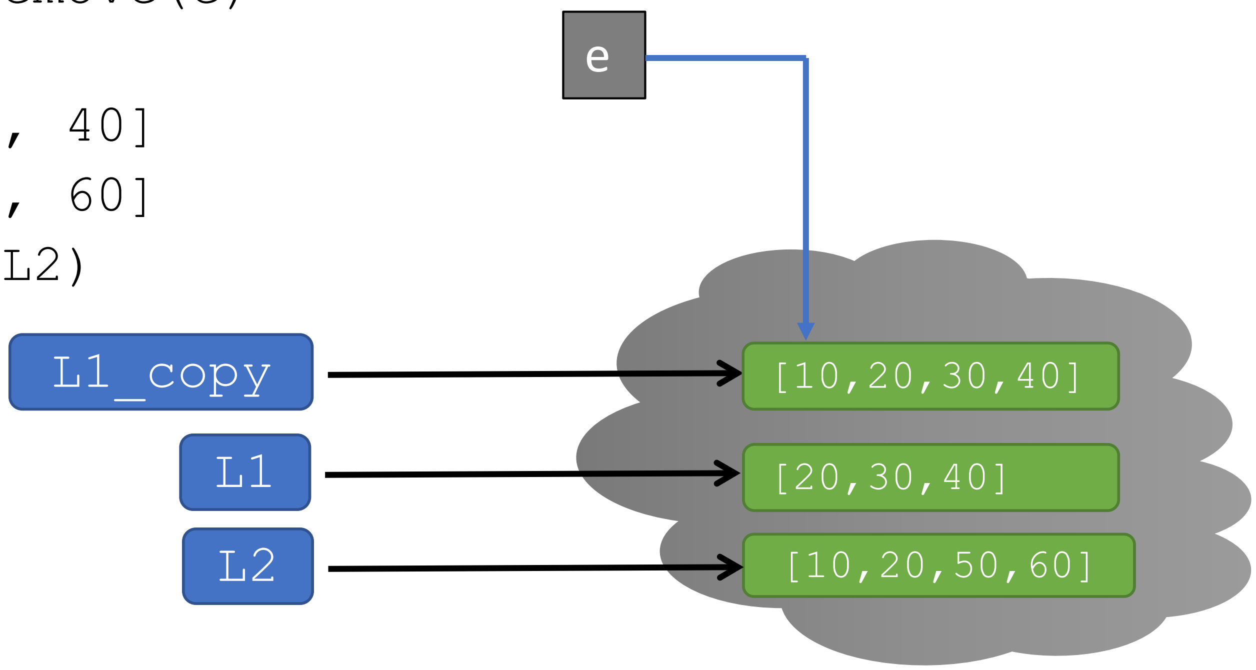
```
L1 = [10, 20, 30, 40]
L2 = [10, 20, 50, 60]
remove_dups(L1, L2)
```

- New version works!
    - Iterate over a copy
    - Mutate original list, not the copy
    - Indexing is now consistent

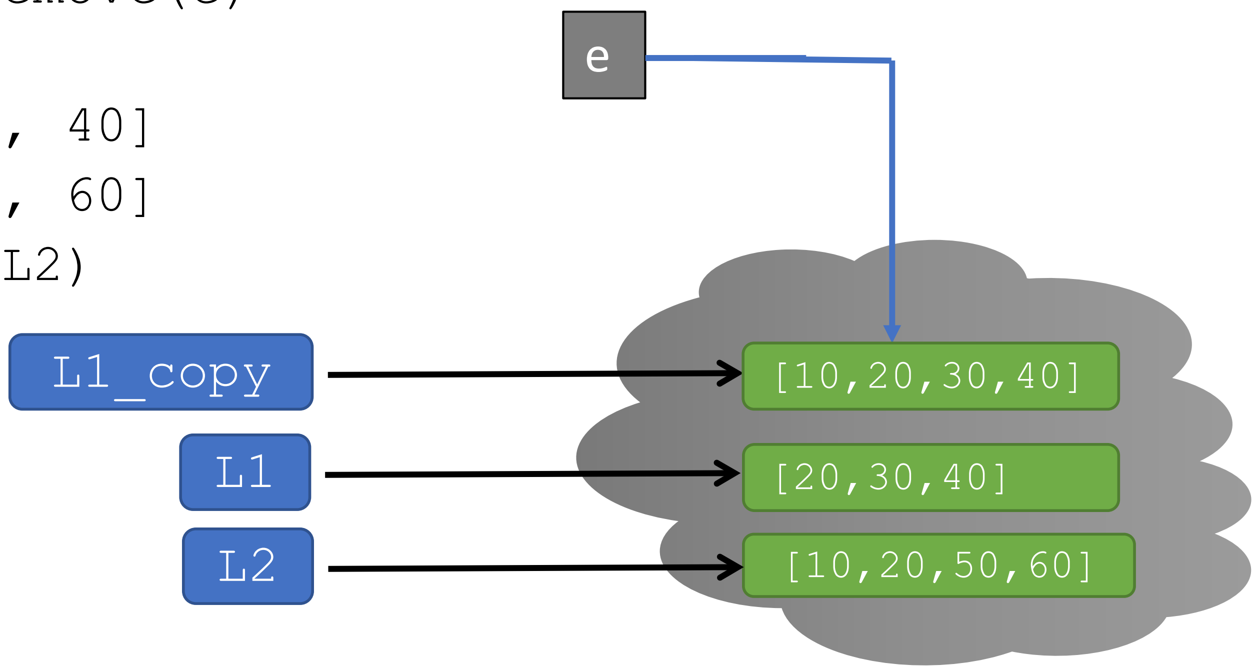
```
def remove_dups(L1, L2):
    L1_copy = L1[:]
    for e in L1_copy:
        if e in L2:
            L1.remove(e)

L1 = [10, 20, 30, 40]
L2 = [10, 20, 50, 60]
remove_dups(L1, L2)
```



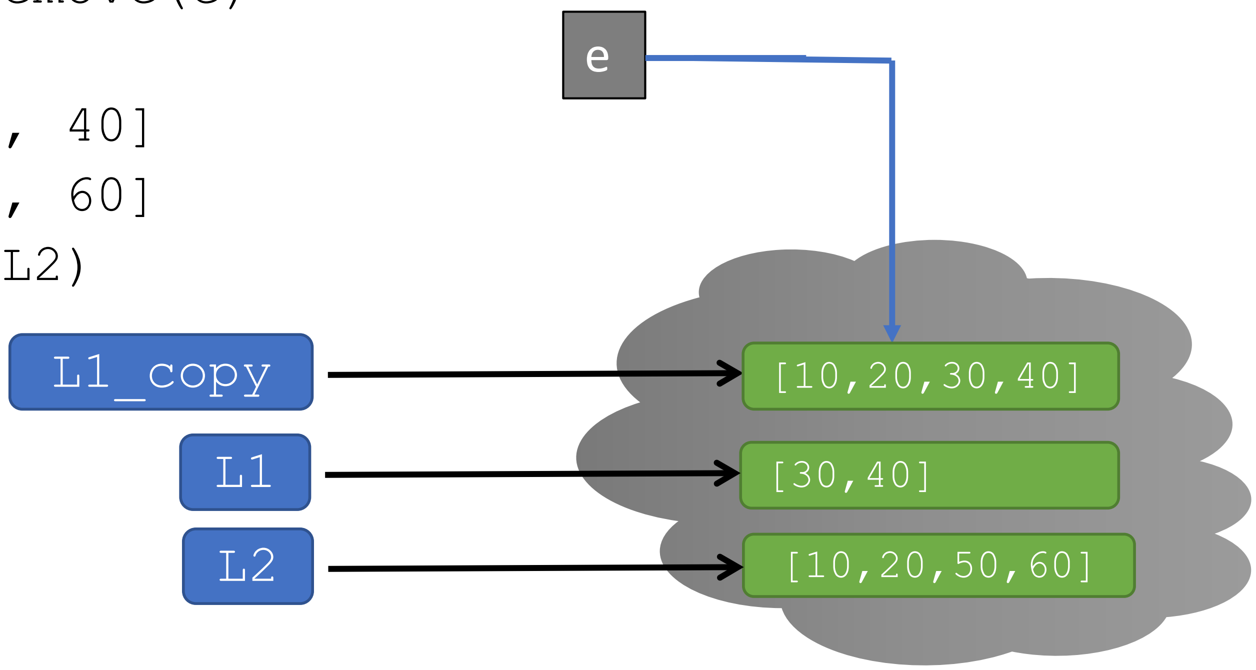
```
def remove_dups(L1, L2):
    L1_copy = L1[:]
    for e in L1_copy:
        if e in L2:
            L1.remove(e)

L1 = [10, 20, 30, 40]
L2 = [10, 20, 50, 60]
remove_dups(L1, L2)
```



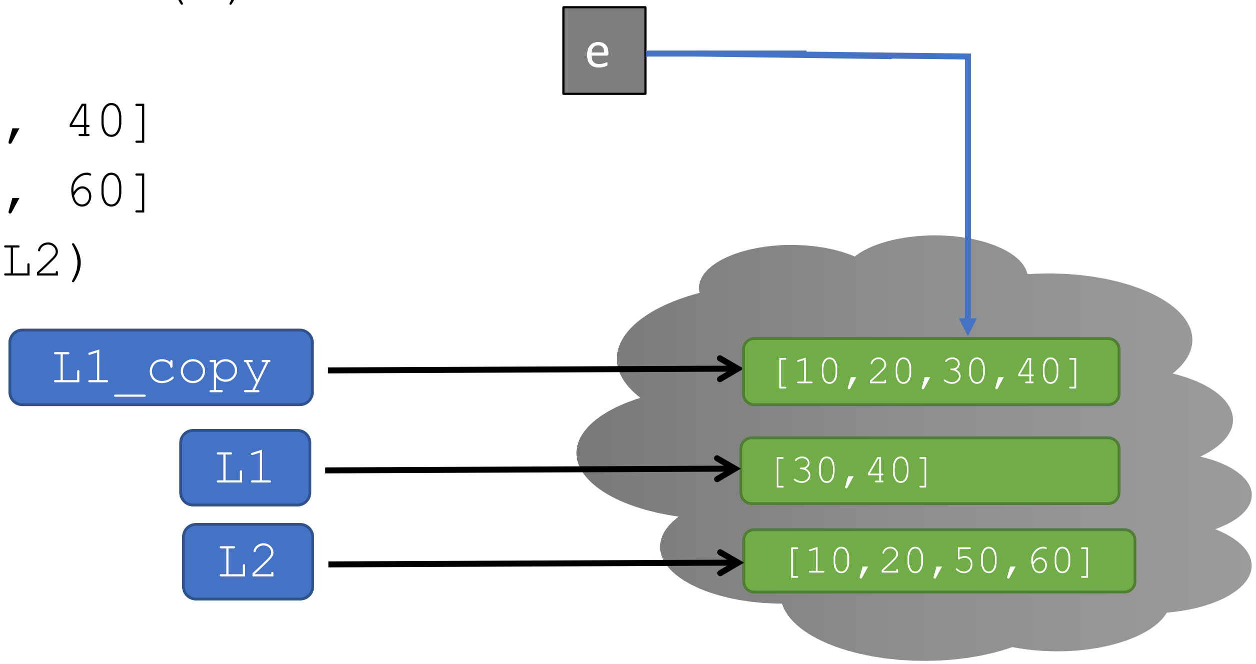
```
def remove_dups(L1, L2):
    L1_copy = L1[:]
    for e in L1_copy:
        if e in L2:
            L1.remove(e)

L1 = [10, 20, 30, 40]
L2 = [10, 20, 50, 60]
remove_dups(L1, L2)
```



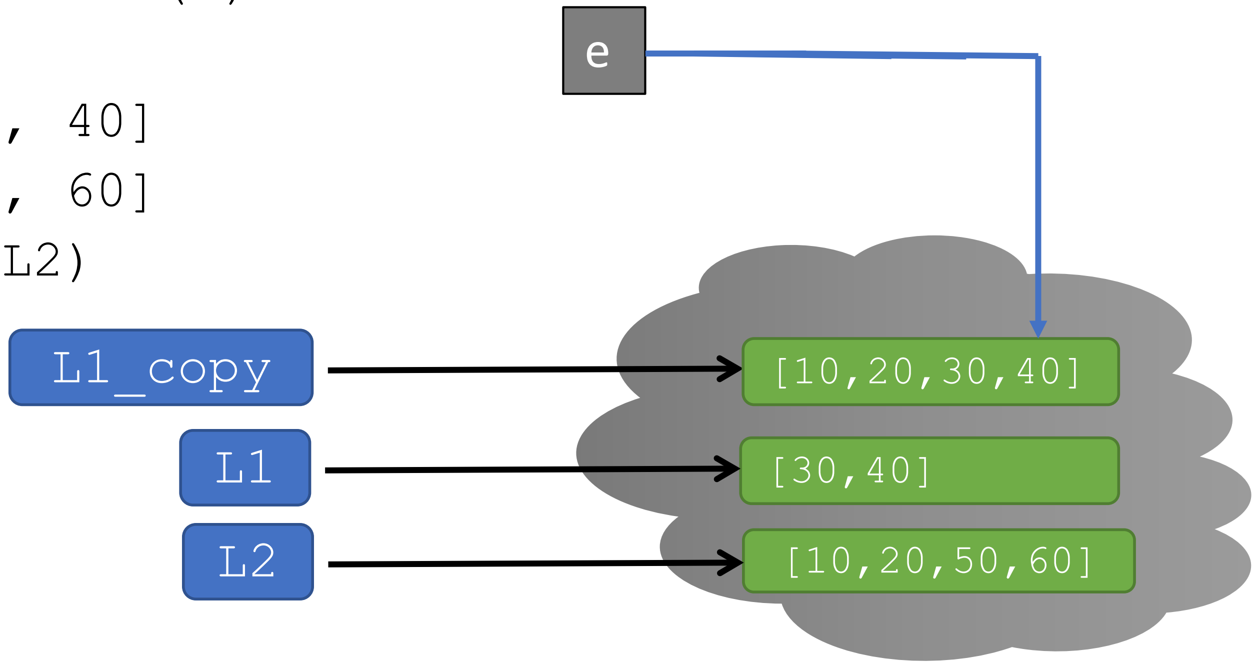
```
def remove_dups(L1, L2):
    L1_copy = L1[:]
    for e in L1_copy:
        if e in L2:
            L1.remove(e)

L1 = [10, 20, 30, 40]
L2 = [10, 20, 50, 60]
remove_dups(L1, L2)
```



```
def remove_dups(L1, L2):
    L1_copy = L1[:]
    for e in L1_copy:
        if e in L2:
            L1.remove(e)

L1 = [10, 20, 30, 40]
L2 = [10, 20, 50, 60]
remove_dups(L1, L2)
```



```
def remove_dups(L1, L2):
    L1_copy = L1[:]
    for e in L1_copy:
        if e in L2:
            L1.remove(e)

L1 = [10, 20, 30, 40]
L2 = [10, 20, 50, 60]
remove_dups(L1, L2)
```



ALIASING

- City may be known by many names
- Attributes of a city
  - Small, tech-savvy
- All nicknames point to the **same city**
  - Add new attribute to **one nickname** …

Boston
The Hub
Beantown
Athens of America

Boston	small	tech-savvy	snowy

… all the **aliases** refer to the old attribute and all the new ones

The Hub	small	tech-savvy	snowy
Beantown	small	tech-savvy	snowy

MUTATION AND ITERATION **WITH ALIAS**

`L1_copy = L1`

- Assignment (= sign) on mutable obj creates an **alias**, not a clone

```
def remove_dups(L1, L2):
    L1_copy = L1
    for e in L1_copy:
        if e in L2:
            L1.remove(e)
```

The same object as L1

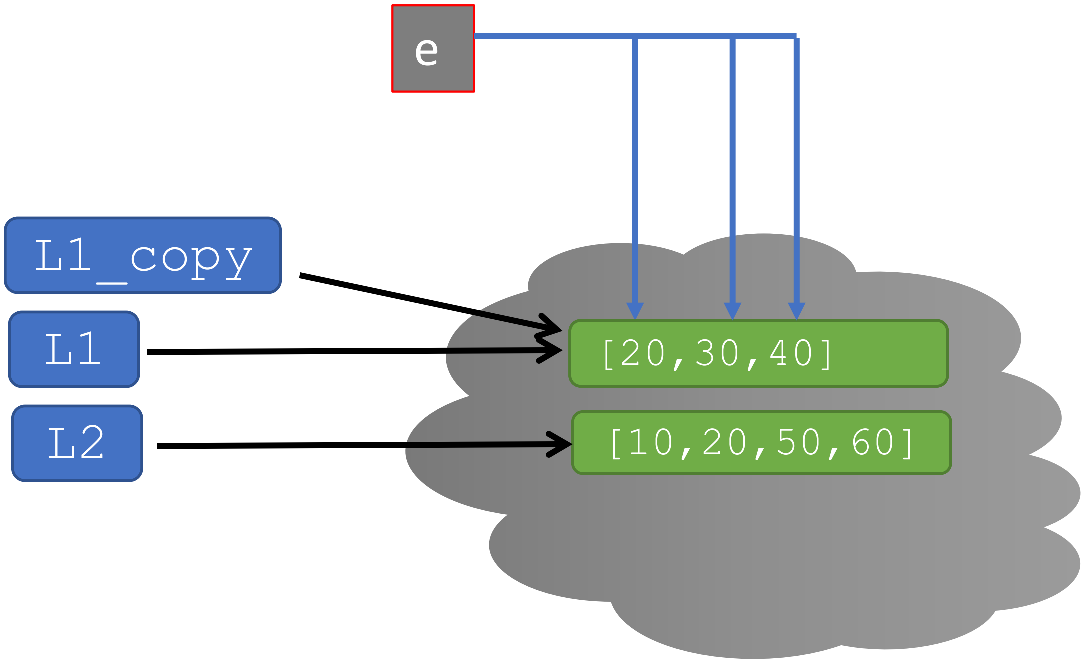
```
def remove_dups(L1, L2):
    L1_copy = L1[:]
    for e in L1_copy:
        if e in L2:
            L1.remove(e)
```

```
L1 = [10, 20, 30, 40]
L2 = [10, 20, 50, 60]
remove_dups(L1, L2)
```

Note that `L1_copy = L1` does NOT clone

- Using a simple assignment without making a copy
  - Makes an alias for list (**same list object referenced by another name**)
  - It's like iterating over L itself, it doesn't work!

```
def remove_dups(L1, L2):
    L1_copy = L1
    for e in L1_copy:
        if e in L2:
            L1.remove(e)

L1 = [10, 20, 30, 40]
L2 = [10, 20, 50, 60]
remove_dups(L1, L2)
```



BIG IDEA

When you pass a list as a parameter to a function, you are making an alias.

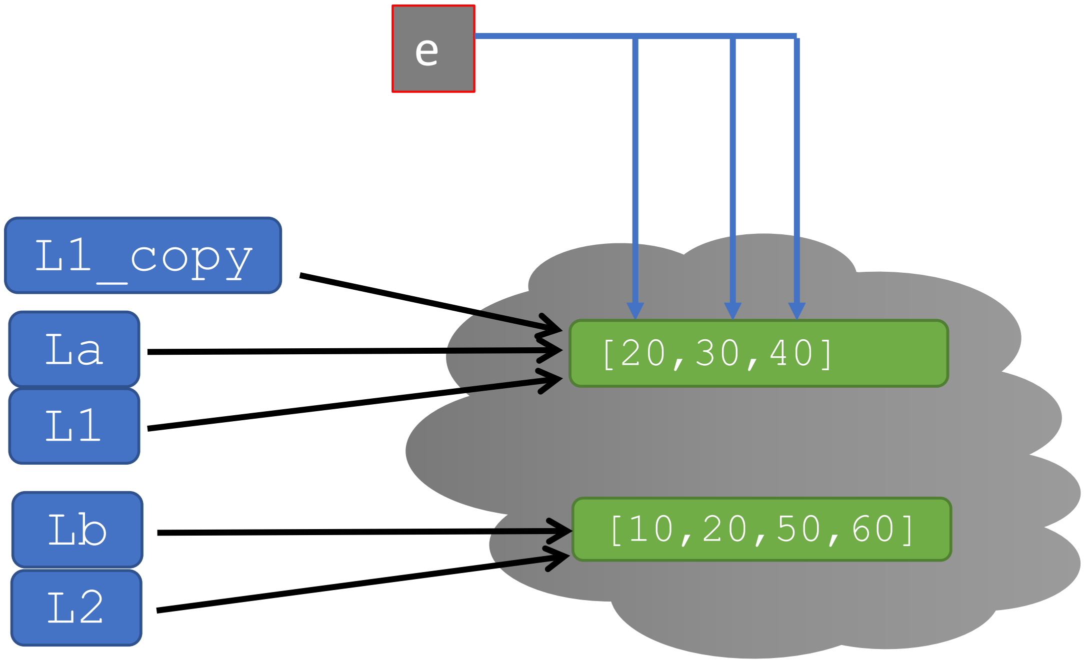
The **actual parameter** (from the function **call**) is an **alias** for the **formal parameter** (from the function **definition**).

Alias for La

Alias for Lb

```
def remove_dups(L1, L2):
    L1_copy = L1
    for e in L1_copy:
        if e in L2:
            L1.remove(e)

La = [10, 20, 30, 40]
Lb = [10, 20, 50, 60]
remove_dups(La, Lb)
print(La)
```

L1 was mutated, but it's an alias for La



ALIASES, SHALLOW COPIES, AND DEEP COPIES WITH MUTABLE ELEMENTS

CONTROL COPYING

- Assignment just creates a new pointer to same object

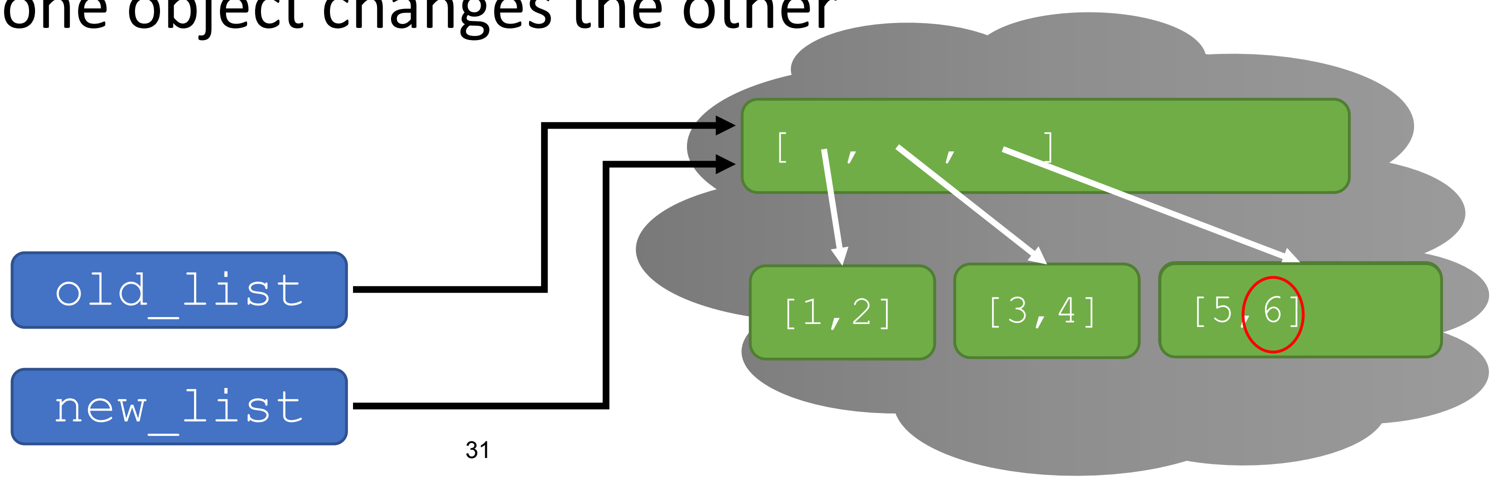
```
old_list = [[1,2],[3,4],[5,'foo']]
new_list = old_list
```

Assignment creates an alias for an existing data structure

```
new_list[2][1] = 6
print("New list:", new_list)
print("Old list:", old_list)
```

```
New list: [[1,2],[3,4],[5,6]]
Old list: [[1,2],[3,4],[5,6]]
```

- So mutating one object changes the other



CONTROL COPYING

- Suppose we want to create a copy of a list, not just a shared pointer
- Shallow copying does this at the **top level of the list**
  - Equivalent to syntax [:]
  - Any mutable elements are NOT copied
- Use this when your list contains immutable objects only

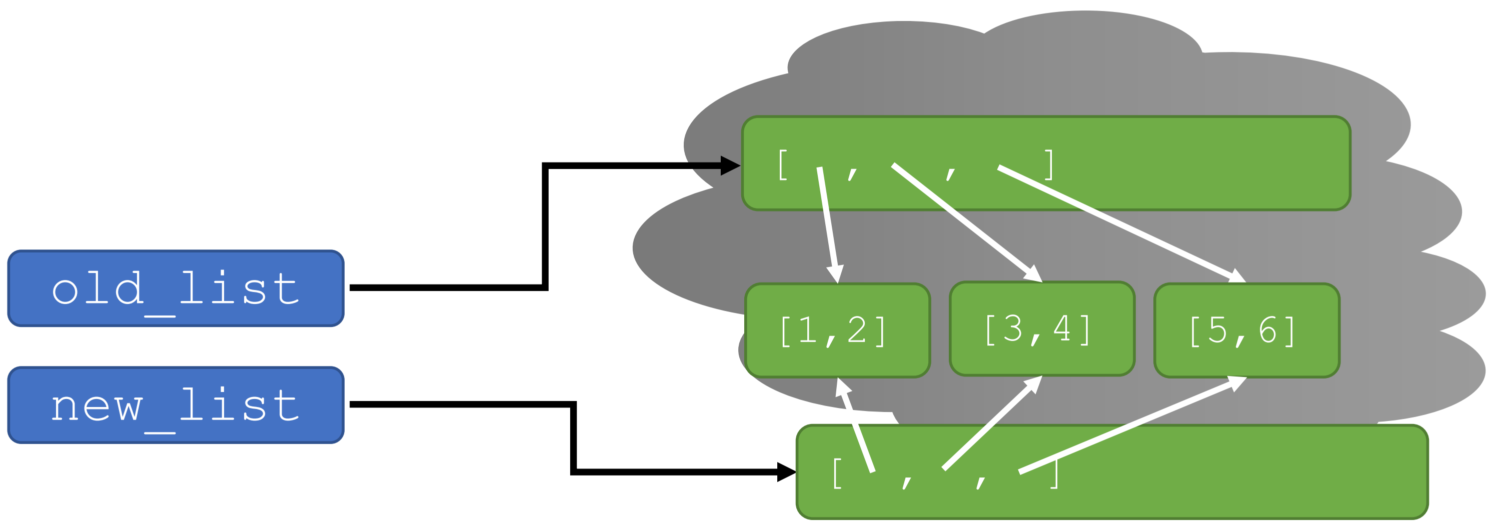
```
import copy
old_list = [[1,2],[3,4],[5,6]]
new_list = copy.copy(old_list)

print("New list:", new_list)
print("Old list:", old_list)
```

```
old_list = [[1,2],[3,4],[5,6]]
new_list = copy.copy(old_list)
```

Copy creates a new data structure, but actual elements are shared

```
print("New list:", new_list)
print("Old list:", old_list)
```

```
New list: [[1,2],[3,4],[5,6]]
Old list: [[1,2],[3,4],[5,6]]
```



CONTROL COPYING

- Now we mutate the top level structure

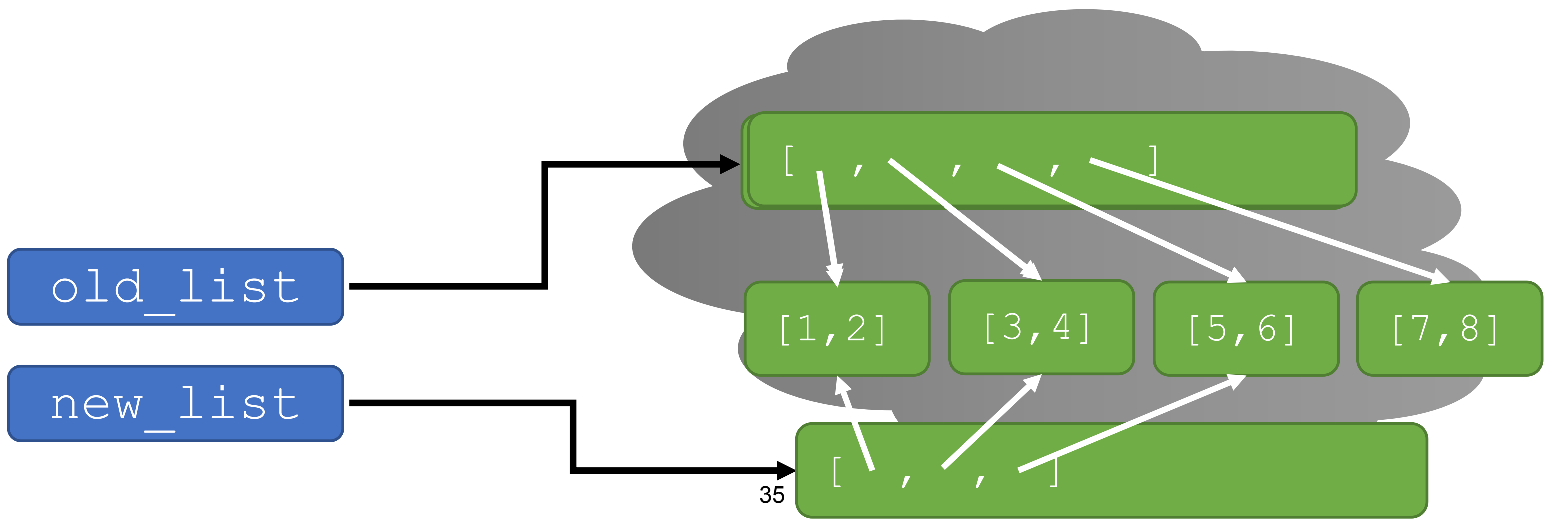
```
import copy
old_list = [[1,2],[3,4],[5,6]]
new_list = copy.copy(old_list)

old_list.append([7,8])
print("New list:", new_list)
print("Old list:", old_list)
```

```
old_list = [[1,2],[3,4],[5,6]]
new_list = copy.copy(old_list)
```

```
old_list.append([7,8])
print("New list:", new_list)
print("Old list:", old_list)
```

Copy creates a new data structure with shared elements, so mutating top level structure of one does not affect the clone

```
New list: [[1,2],[3,4],[5,6]]
Old list: [[1,2],[3,4],[5,6],[7,8]]
```



CONTROL COPYING

- But if we change an element in one of the sub-structures, they are shared!
- If your elements are not mutable then this is not a problem

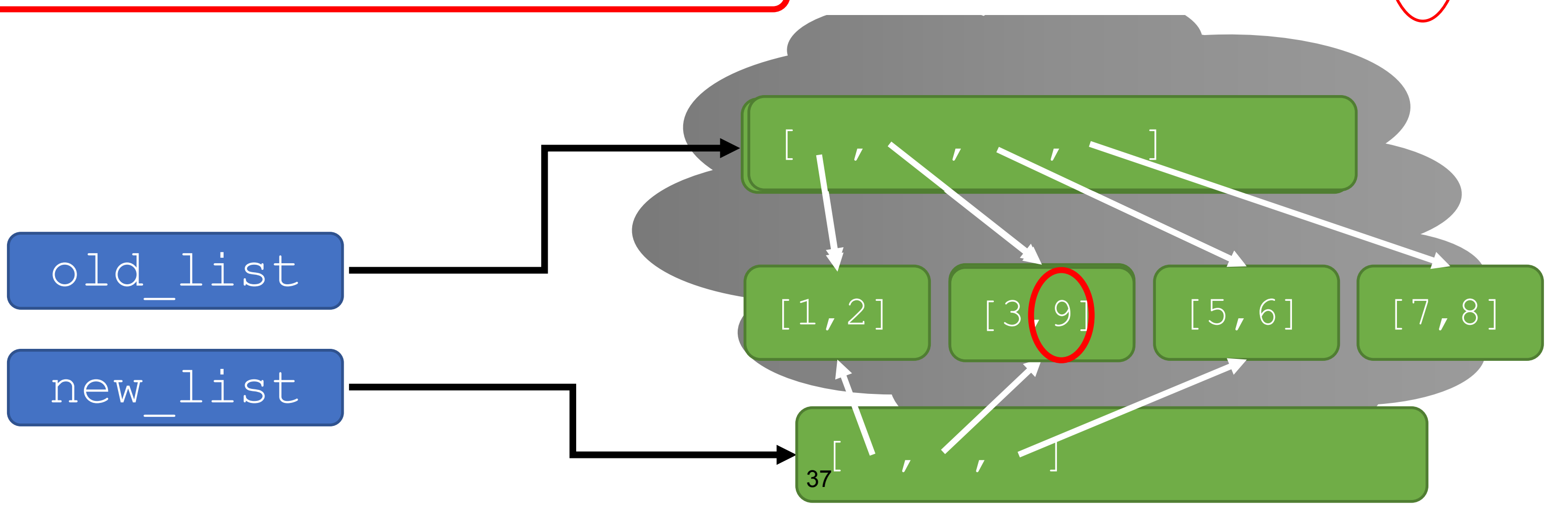
```
import copy
old_list = [[1,2],[3,4],[5,6]]
new_list = copy.copy(old_list)

old_list.append([7,8])
old_list[1][1] = 9
print("New list:", new_list)
print("Old list:", old_list)
```

```python
old_list = [[1,2],[3,4],[5,6]]
new_list = copy.copy(old_list)
```

```python
old_list.append([7,8])
old_list[1][1] = 9
print("New list:", new_list)
print("Old list:", old_list)
```

```
New list: [[1,2],[3,9],[5,6]]

Old list: [[1,2],[3,9],[5,6],[7,8]]
```

Shallow copying creates a new data structure with shared elements; so top level are clones, but elements are aliases; mutating an element will thus affect the other object



CONTROL COPYING

- If we want all structures to be new copies, we need a deep copy
- Use deep copy when your list might have mutable elements to ensure every structure at every level is copied

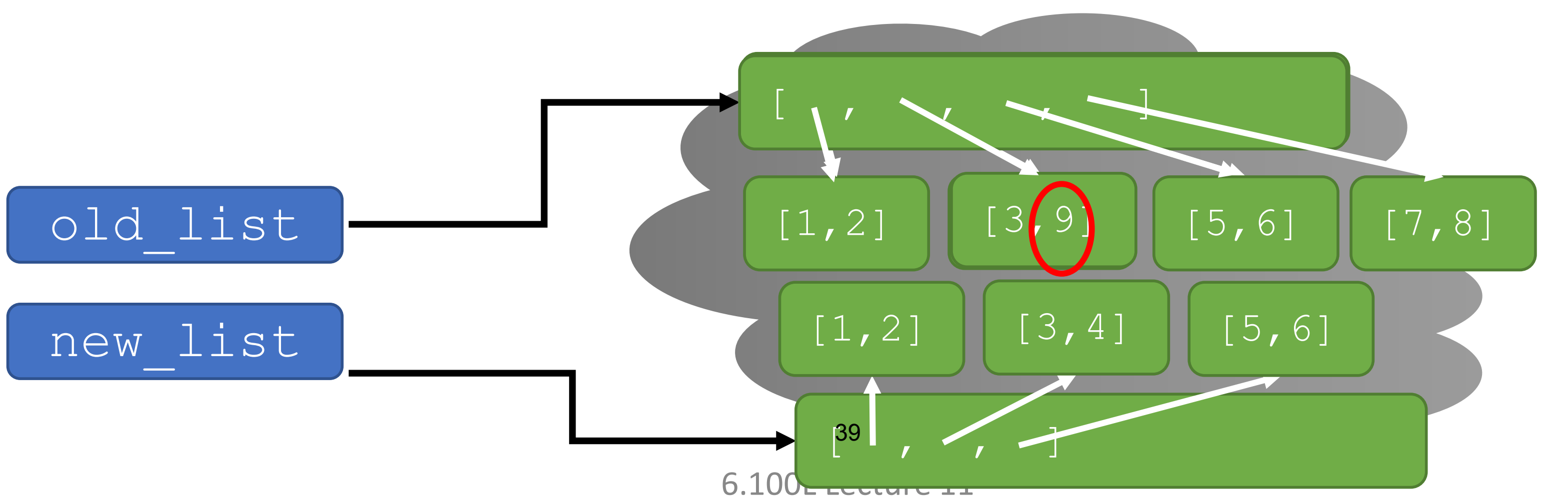
```
import copy
old_list = [[1,2],[3,4],[5,6]]
new_list = copy.deepcopy(old_list)

old_list.append([7,8])
old_list[1][1] = 9
print("New list:", new_list)
print("Old list:", old_list)
```

```
old_list = [[1,2],[3,4],[5,6]]
new_list = copy.deepcopy(old_list)

old_list.append([7,8])
old_list[1][1] = 9
print("New list:", new_list)
print("Old list:", old_list)
```

New list: [[1,2],[3,4],[5,6]]

Old list: [[1,2],[3,9],[5,6],[7,8]]

Deep copying creates clones at all levels of structure; thus mutating one does not affect the other at any level



LISTS in MEMORY

- Separate the idea of the **object vs. the name we give** an object
  - A list is an object in memory
  - Variable name points to object
- Lists are **mutable** and behave differently than immutable types
- Using **equal sign** between mutable objects **creates aliases**
  - Both variables point to the same object in memory
  - Any variable pointing to that object is affected by mutation of object, even if mutation is by referencing another name
- If you want a copy, you explicitly tell Python to make a copy
- Key phrase to keep in mind when working with lists is **side effects,** especially when dealing with **aliases** – two names pointing to the same structure in memory
- ***Python Tutor is your best friend to help sort this out!*** http://www.pythontutor.com/

WHY LISTS and TUPLES?

- If mutation can cause so many problems, why do we even want to have lists, **why not just use tuples**?
  - Efficiency – if processing very large sequences, don't want to have to copy every time we change an element

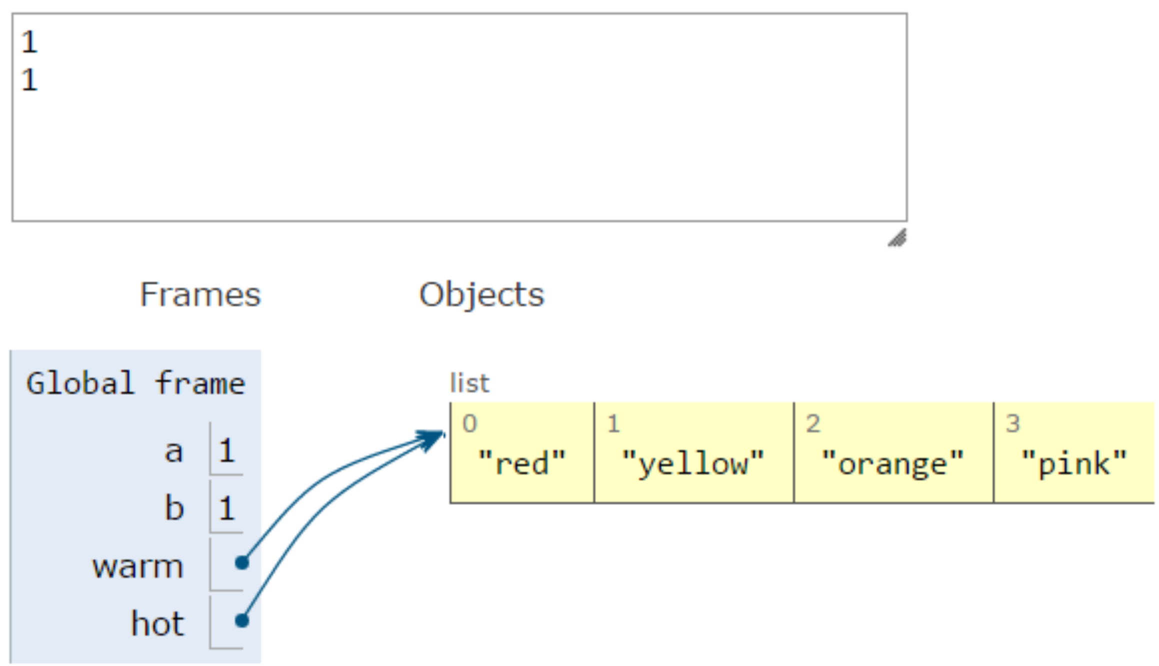
- If lists basically do everything that tuples do, **why not just have lists**?
  - Immutable structures can be very valuable in context of other object types
  - Don't want to accidentally have other code mutate some important data, tuples safeguard against this
  - They can be a bit faster

AT HOME TRACING EXAMPLES SHOWCASING ALIASING AND CLONING

ALIASES

- `hot` is an **alias** for `warm` – changing one changes the other!
- `append()` has a side effect

```python
a = 1
b = a
print(a)
print(b)

warm = ['red', 'yellow', 'orange']
hot = warm


```



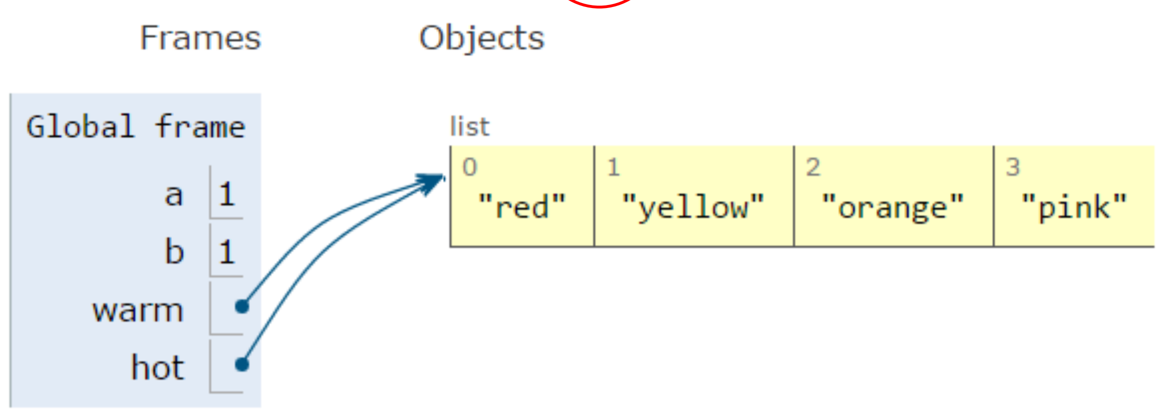
ALIASES

- `hot` is an **alias** for `warm` – changing one changes the other!
- `append()` has a side effect

Never explicitly changed warm, but its value has changed

```
a = 1
b = a
print(a)
print(b)

warm = ['red', 'yellow', 'orange']
hot = warm
hot.append('pink')
print(hot)
print(warm)
```

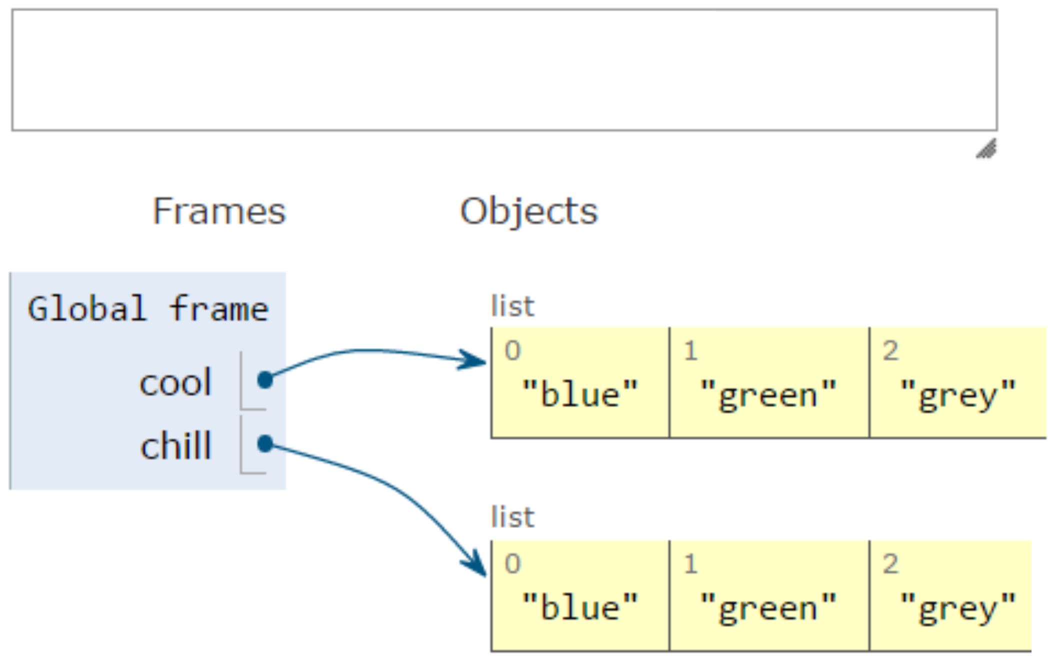
```
1
1
['red', 'yellow', 'orange', 'pink']
['red', 'yellow', 'orange', 'pink']
```



CLONING A LIST

- Create a new list and **copy every element** using a clone
  `chill = cool[:]`

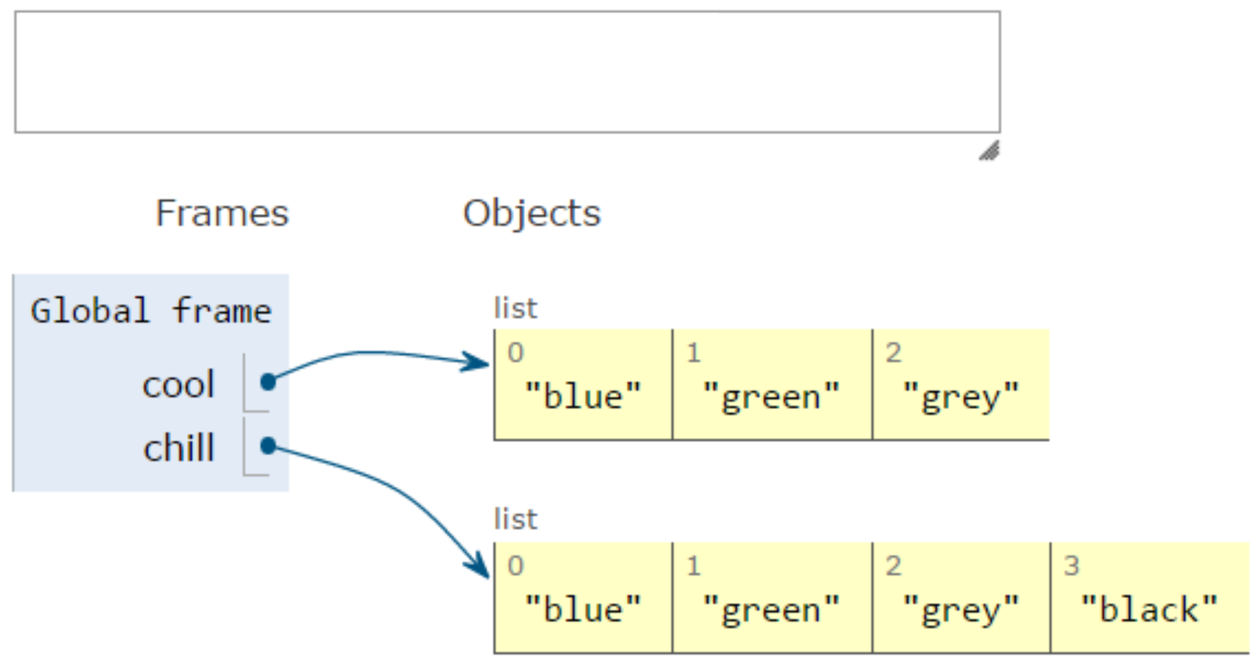
```
cool = ['blue', 'green', 'grey']
```



CLONING A LIST

- Create a new list and **copy every element** using a clone
  `chill = cool[:]`

```
cool = ['blue', 'green', 'grey']
chill = cool[:]
chill.append('black')
print(chill)
print(cool)
```



CLONING A LIST

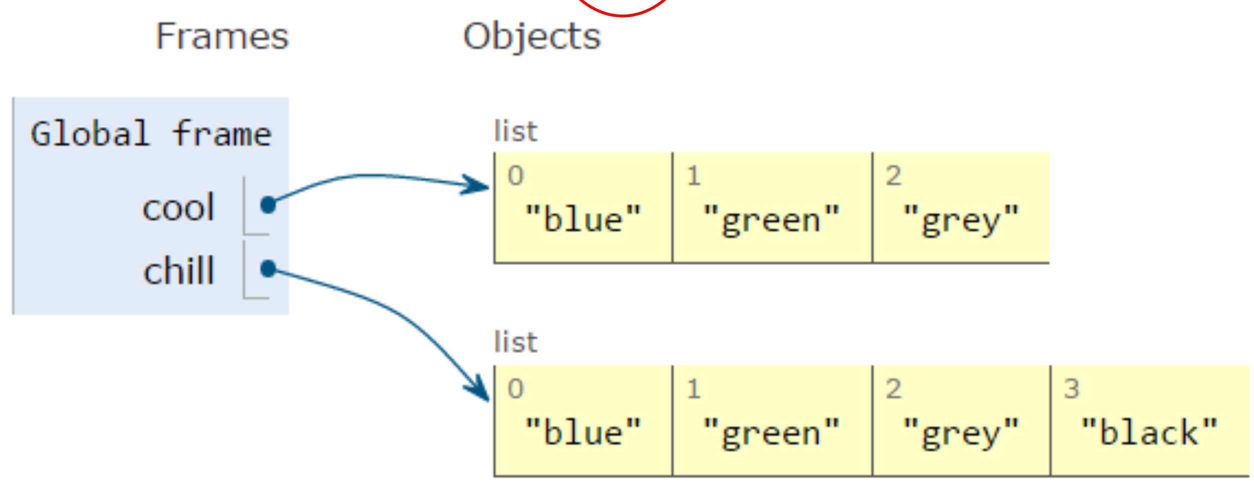
- Create a new list and **copy every element** using a clone

```
chill = cool[:]
```

Because chill is a clone, changing chill does not change cool

```
cool = ['blue', 'green', 'grey']
chill = cool[:]
chill.append('black')
print(chill)
print(cool)
```

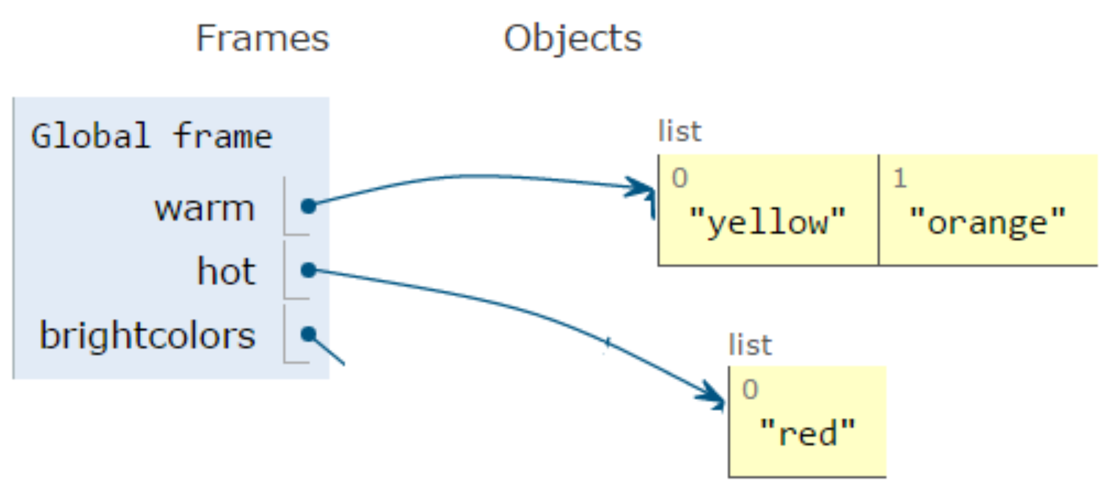
```
['blue', 'green', 'grey', 'black']
['blue', 'green', 'grey']
```



LISTS OF LISTS OF LISTS OF….

- Can have **nested** lists
- Side effects still possible after mutation

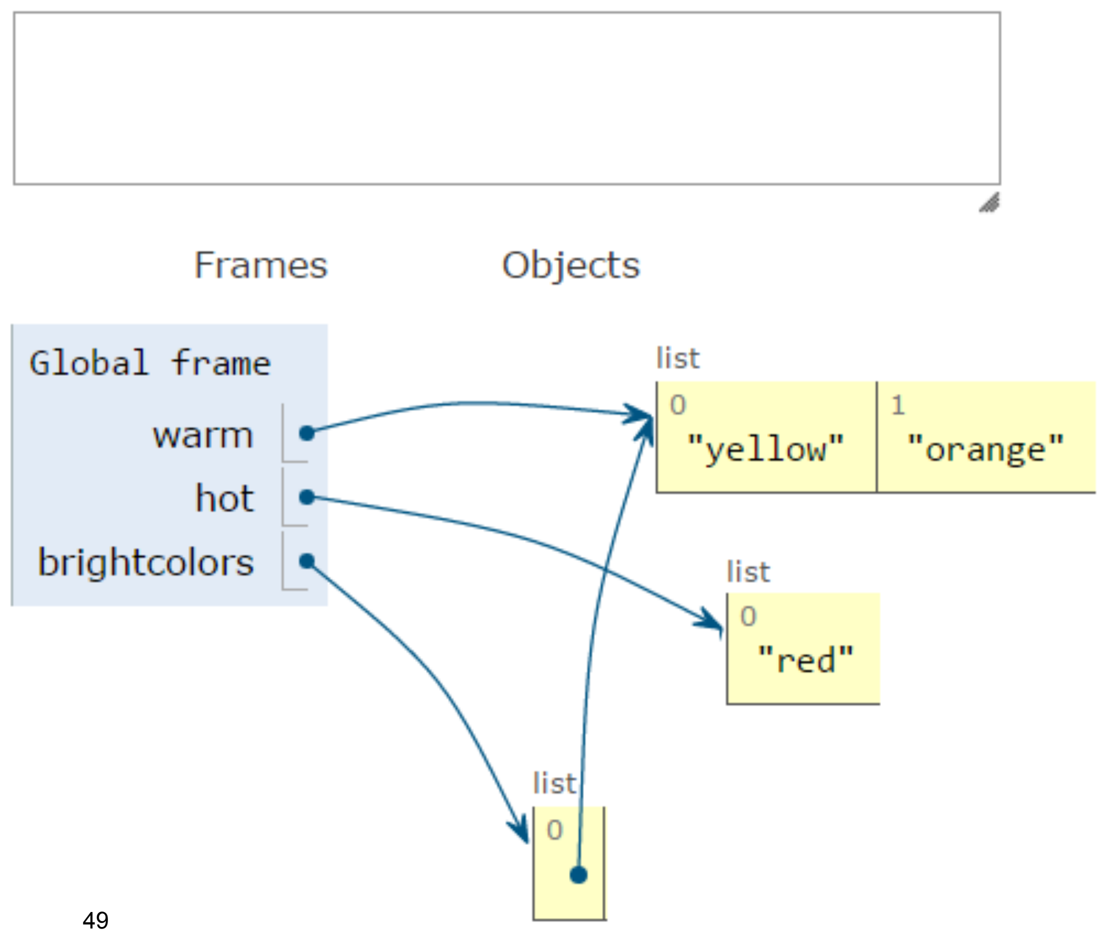
```
warm = ['yellow', 'orange']
hot = ['red']
brightcolors = [warm]
brightcolors.append(hot)
print(brightcolors)
```



LISTS OF LISTS OF LISTS OF….

- Can have **nested** lists
- Side effects still possible after mutation

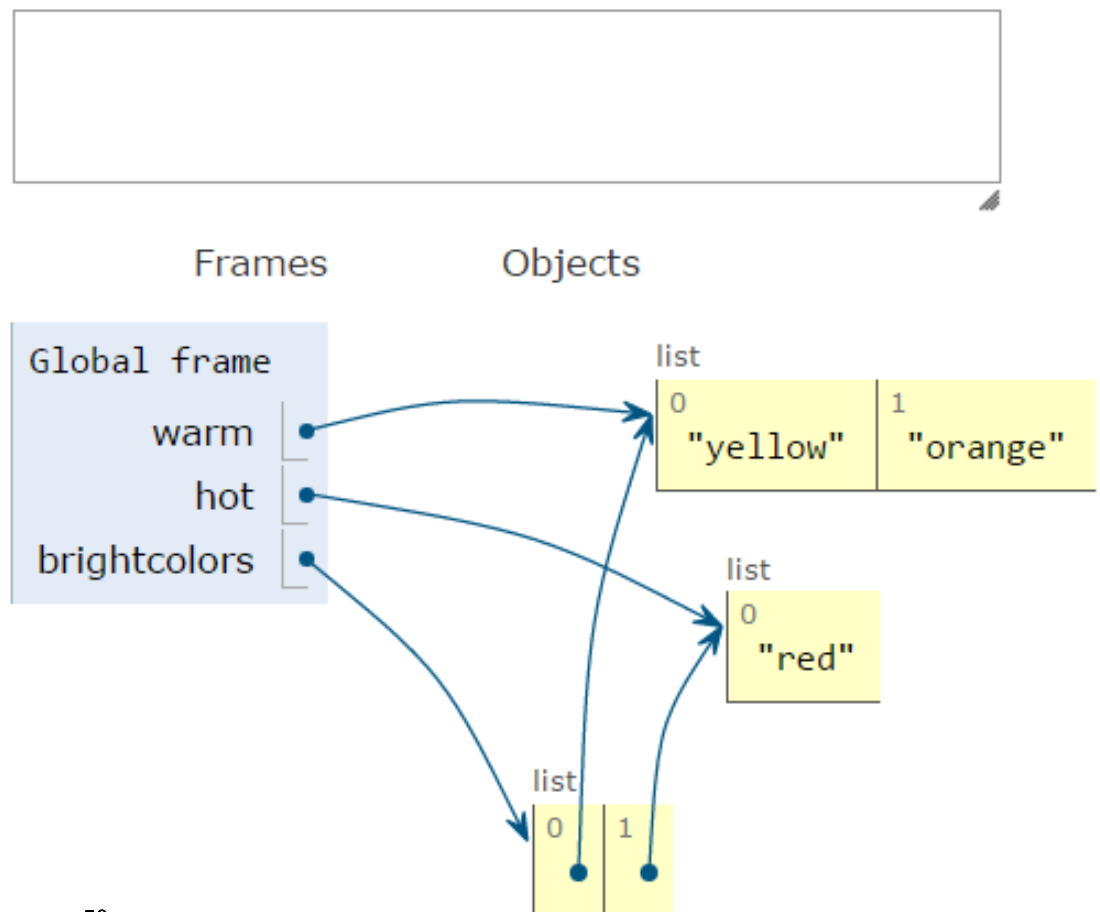
```
warm = ['yellow', 'orange']
hot = ['red']
brightcolors = [warm]
brightcolors.append(hot)
print(brightcolors)
```



LISTS OF LISTS OF LISTS OF….

- Can have **nested** lists
- Side effects still possible after mutation

```
warm = ['yellow', 'orange']
hot = ['red']
brightcolors = [warm]
brightcolors.append(hot)
print(brightcolors)
```

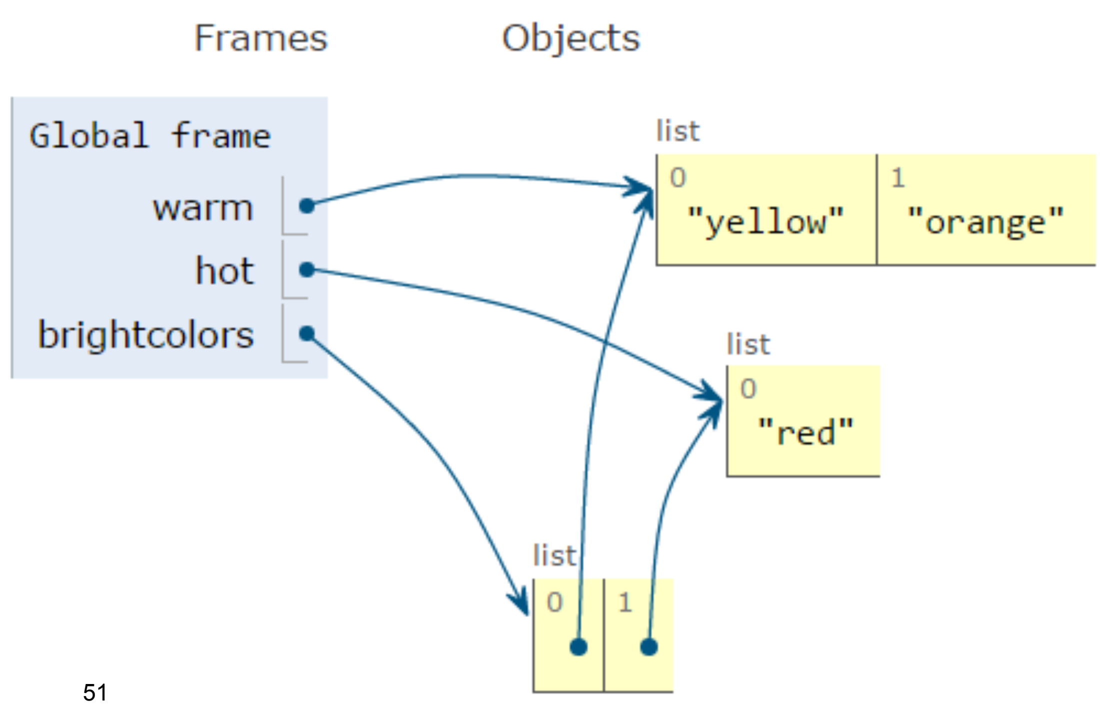


LISTS OF LISTS OF LISTS OF….

- Can have **nested** lists
- Side effects still possible after mutation

```
warm = ['yellow', 'orange']
hot = ['red']
brightcolors = [warm]
brightcolors.append(hot)
print(brightcolors)
```

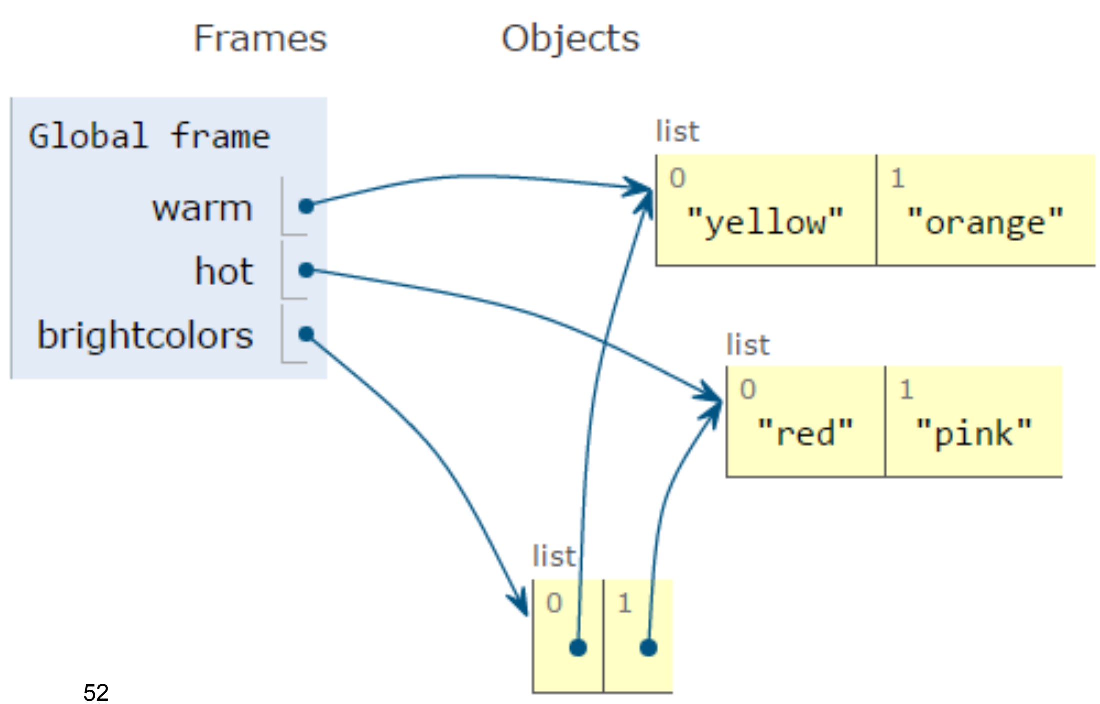
```
[['yellow', 'orange'], ['red']]
```



LISTS OF LISTS OF LISTS OF….

- Can have **nested** lists
- Side effects still possible after mutation

```
[['yellow', 'orange'], ['red']]
```



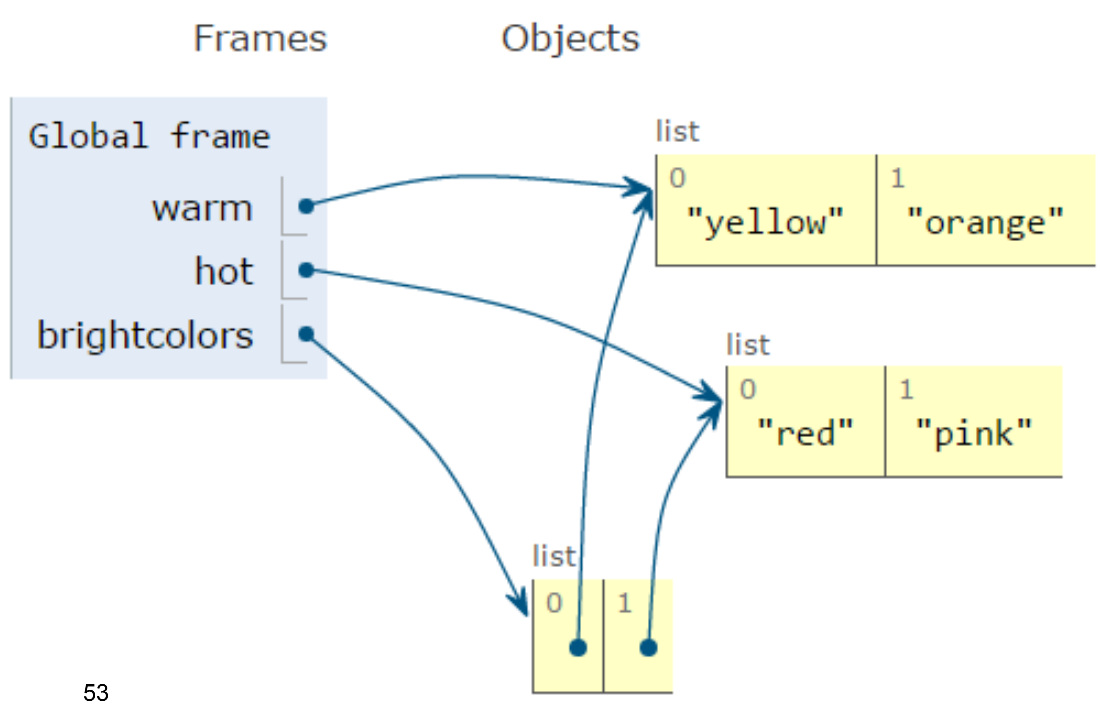

```python
warm = ['yellow', 'orange']
hot = ['red']
brightcolors = [warm]
brightcolors.append(hot)
print(brightcolors)
hot.append('pink')
print(hot)
print(brightcolors)
```

LISTS OF LISTS OF LISTS OF….

- Can have **nested** lists
- Side effects still possible after mutation

```
[['yellow', 'orange'], ['red']]
['red', 'pink']
[['yellow', 'orange'], ['red', 'pink']]
```

```
warm = ['yellow', 'orange']
hot = ['red']
brightcolors = [warm]
brightcolors.append(hot)
print(brightcolors)
hot.append('pink')
print(hot)
print(brightcolors)
```



LIST COMPREHENSION, FUNCTIONS AS OBJECTS, TESTING, DEBUGGING

(download slides and .py files to follow along)

6.100L Lecture 12

Ana Bell

LIST COMPREHENSIONS

LIST COMPREHENSIONS

- Applying a **function to every element of a sequence**, then creating a new list with these values is a common concept
- Example:

New list

Function to apply

```
def f(L):
    Lnew = []
    for e in L:
        Lnew.append(e**2)
    return Lnew
```

- Python provides a concise one-liner way to do this, called a **list comprehension**
  - Creates a new list
  - Applies a function to every element of another iterable
  - Optional, only apply to elements that satisfy a test

```
[expression for elem in iterable if test]
```

LIST COMPREHENSIONS

- Create a **new** list, by applying a function to every element of another iterable that satisfies a test

```
def f(L):
  Lnew = []
  for e in L:
    Lnew.append(e**2)
  return Lnew
```

New list

Look at every element

Function to apply

```
Lnew = [e**2 for e in L]
```

New list

LIST COMPREHENSIONS

- Create a **new** list, by applying a function to every element of another iterable that satisfies a test

```
def f(L):
  Lnew = []
  for e in L:
    Lnew.append(e**2)
  return Lnew
```

```
Lnew = [e**2 for e in L]
```

New list

Function to apply only if test is true

```
def f(L):
  Lnew = []
  for e in L:
    if e%2==0:
      Lnew.append(e**2)
  return Lnew
```

LIST COMPREHENSIONS

- Create a **new** list, by applying a function to every element of another iterable that satisfies a test

```
def f(L):
  Lnew = []
  for e in L:
    Lnew.append(e**2)
  return Lnew
```

```
Lnew = [e**2 for e in L]
```

```
def f(L):
  Lnew = []
  for e in L:
    if e%2==0:
      Lnew.append(e**2)
  return Lnew
```

New list

```
Lnew = [e**2 for e in L if e%2==0]
```

LIST COMPREHENSIONS

- Create a **new** list, by applying a function to every element of another iterable that satisfies a test

```
def f(L):
  Lnew = []
  for e in L:
    Lnew.append(e**2)
  return Lnew
```

```
Lnew = [e**2 for e in L]
```

```
def f(L):
  Lnew = []
  for e in L:
    if e%2==0:
      Lnew.append(e**2)
  return Lnew
```

Loop over elements

```
Lnew = [e**2 for e in L if e%2==0]
```

LIST COMPREHENSIONS

- Create a **new** list, by applying a function to every element of another iterable that satisfies a test

```
def f(L):
  Lnew = []
  for e in L:
    Lnew.append(e**2)
  return Lnew
```

```
Lnew = [e**2 for e in L]
```

```
def f(L):
  Lnew = []
  for e in L:
    if e%2==0:
      Lnew.append(e**2)
  return Lnew
```

Function to apply only if test is true

```
Lnew = [e**2 for e in L if e%2==0]
```

LIST COMPREHENSIONS

[*expression* for *elem* in *iterable* if *test*]

- This is equivalent to invoking this function (where expression is a function that computes that expression)

```
def f(expr, old_list, test = lambda x: True):
    new_list = []
    for e in old_list:
        if test(e):
            new_list.append(expr(e))
    return new_list
```

```
[e**2 for e in range(6)]            → [0, 1, 4, 9, 16, 25]
[e**2 for e in range(8) if e%2 == 0] → [0, 4, 16, 36]
[[e,e**2] for e in range(4) if e%2 != 0] → [[1,1], [3,9]]
```

YOU TRY IT!

- What is the value returned by this expression?
  - Step1: what are **all values** in the sequence
  - Step2: which **subset of values** does the condition filter out?
  - Step3: **apply the function** to those values

```
[len(x) for x in ['xy', 'abcd', 7, '4.0'] if type(x) == str]
```

FUNCTIONS: DEFAULT PARAMETERS

SQUARE ROOT with BISECTION

```
def bisection_root(x):
    epsilon = 0.01
    low = 0
    high = x
    guess = (high + low)/2.0
    while abs(guess**2 - x) >= epsilon:
        if guess**2 < x:
            low = guess
        else:
            high = guess
        guess = (high + low)/2.0
    return guess

print(bisection_root(123))
```

ANOTHER PARAMETER

- Motivation: want a more accurate answer
  `def bisection_root(x)` can be improved
- Options?
  - Change epsilon **inside function** (all function calls are affected)
  - Use an epsilon **outside function** (global variables are bad)
  - Add epsilon as **an argument** to the function

epsilon as a PARAMETER

```
def bisection_root(x, epsilon):
    low = 0
    high = x
    guess = (high + low)/2.0
    while abs(guess**2 - x) >= epsilon:
        if guess**2 < x:
            low = guess
        else:
            high = guess
        guess = (high + low)/2.0
    return guess

print(bisection_root(123, 0.01))
```

KEYWORD PARAMETERS & DEFAULT VALUES

`def bisection_root(x, epsilon)` can be improved

- We added epsilon as an argument to the function
  - **Most of the time** we want some **standard value**, 0.01
  - **Sometimes**, we may want to use some **other value**
- Use a keyword parameter aka a **default parameter**

Epsilon as a KEYWORD PARAMETER

```
def bisection_root(x, epsilon=0.01):
    low = 0
    high = x
    guess = (high + low)/2.0
    while abs(guess**2 - x) >= epsilon:
        if guess**2 < x:
            low = guess
        else:
            high = guess
        guess = (high + low)/2.0
    return guess

print(bisection_root(123))
print(bisection_root(123, 0.5))
```

Default parameter, with default value of 0.01

Uses epsilon as 0.01 (the default one in function def)

Uses epsilon as 0.5

RULES for KEYWORD PARAMETERS

- In the **function definition**:
  - Default parameters must go at the end

- These are **ok for calling a function**:
  - `bisection_root_new(123)`
  - `bisection_root_new(123, 0.001)`
  - `bisection_root_new(123, epsilon=0.001)`
  - `bisection_root_new(x=123, epsilon=0.1)`
  - `bisection_root_new(epsilon=0.1, x=123)`
- These are **not ok for calling a function**:
  - `bisection_root_new(epsilon=0.001, 123) #error`
  - `bisection_root_new(0.001, 123) #no error but wrong`

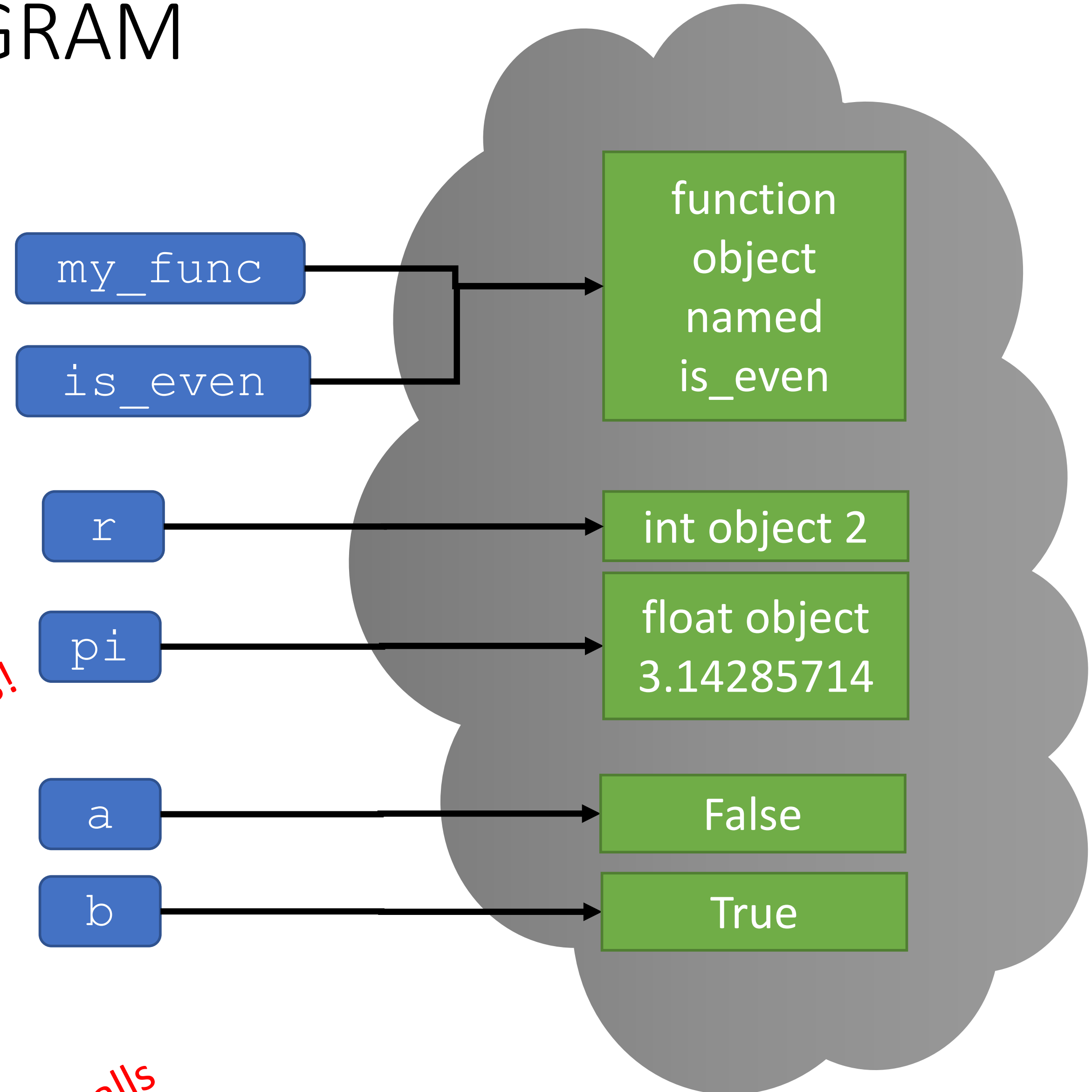
FUNCTIONS RETURNING FUNCTIONS

OBJECTS IN A PROGRAM

```
def is_even(i):
    return i%2 == 0

r = 2

pi = 22/7

my_func = is_even

a = is_even(3)

b = my_func(4)
```

NOT a function call, just names!

Function calls



FUNCTIONS CAN RETURN FUNCTIONS

```
def make_prod(a):
    def g(b):
        return a*b
    return g
```

This function def is inside another function.

This is NOT a function call!

```
val = make_prod(2)(3)
print(val)
```

SAME

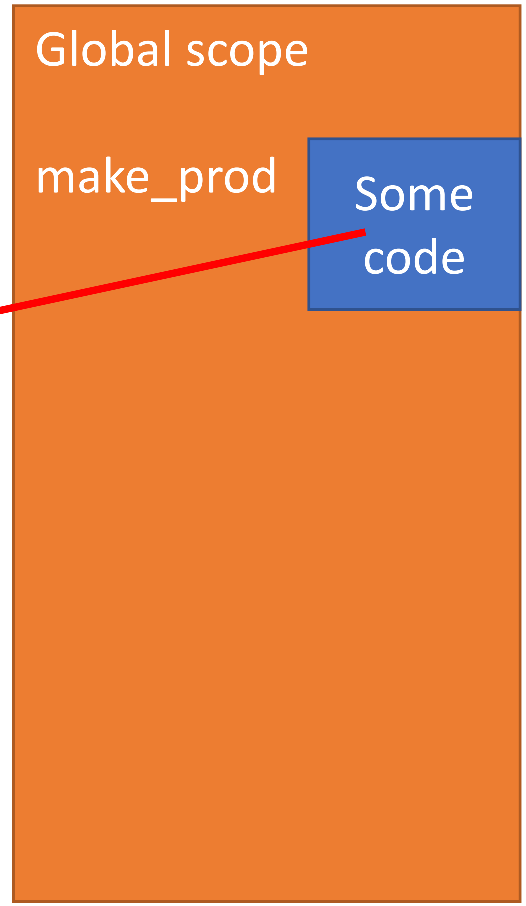
```
doubler = make_prod(2)
val = doubler(3)
print(val)
```

SCOPE DETAILS FOR WAY 1

```
def make_prod(a):
    def g(b):
        return a*b
    return g

val = make_prod(2)(3)
print(val)
```

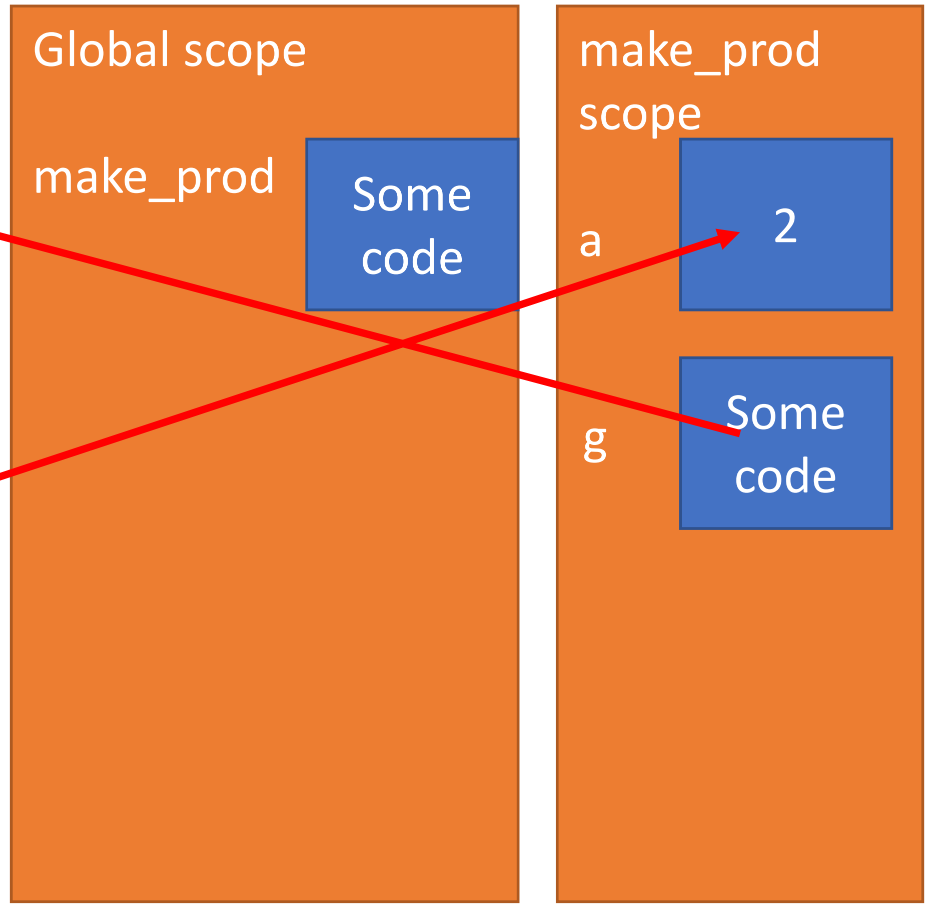
SCOPE DETAILS FOR WAY 1

```
def make_prod(a):
    def g(b):
        return a*b
    return g

val = make_prod(2)(3)
print(val)
```



SCOPE DETAILS FOR WAY 1

```
def make_prod(a):
    def g(b):
        return a*b
    return g

val = make_prod(2)(3)
print(val)
```



NOTE: definition of g is done within scope of make_prod, so binding of g is within that frame/scope

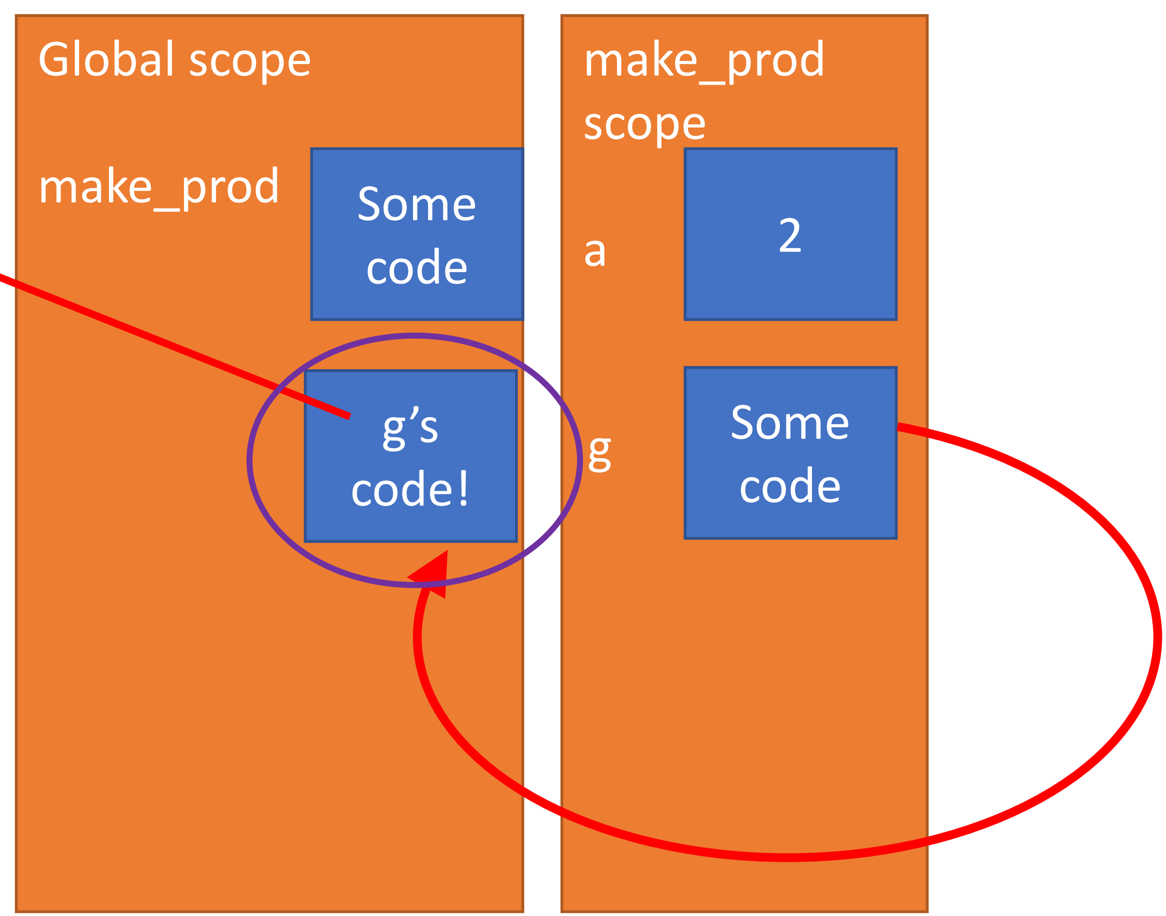
Since g is bound in this frame, cannot access it by evaluation in global frame

g can only be accessed within call to make_prod, and each call will create a new, internal g

SCOPE DETAILS FOR WAY 1

```
def make_prod(a):
    def g(b):
        return a*b
    return g

val = make_prod(2)(3)
print(val)
```

This is g



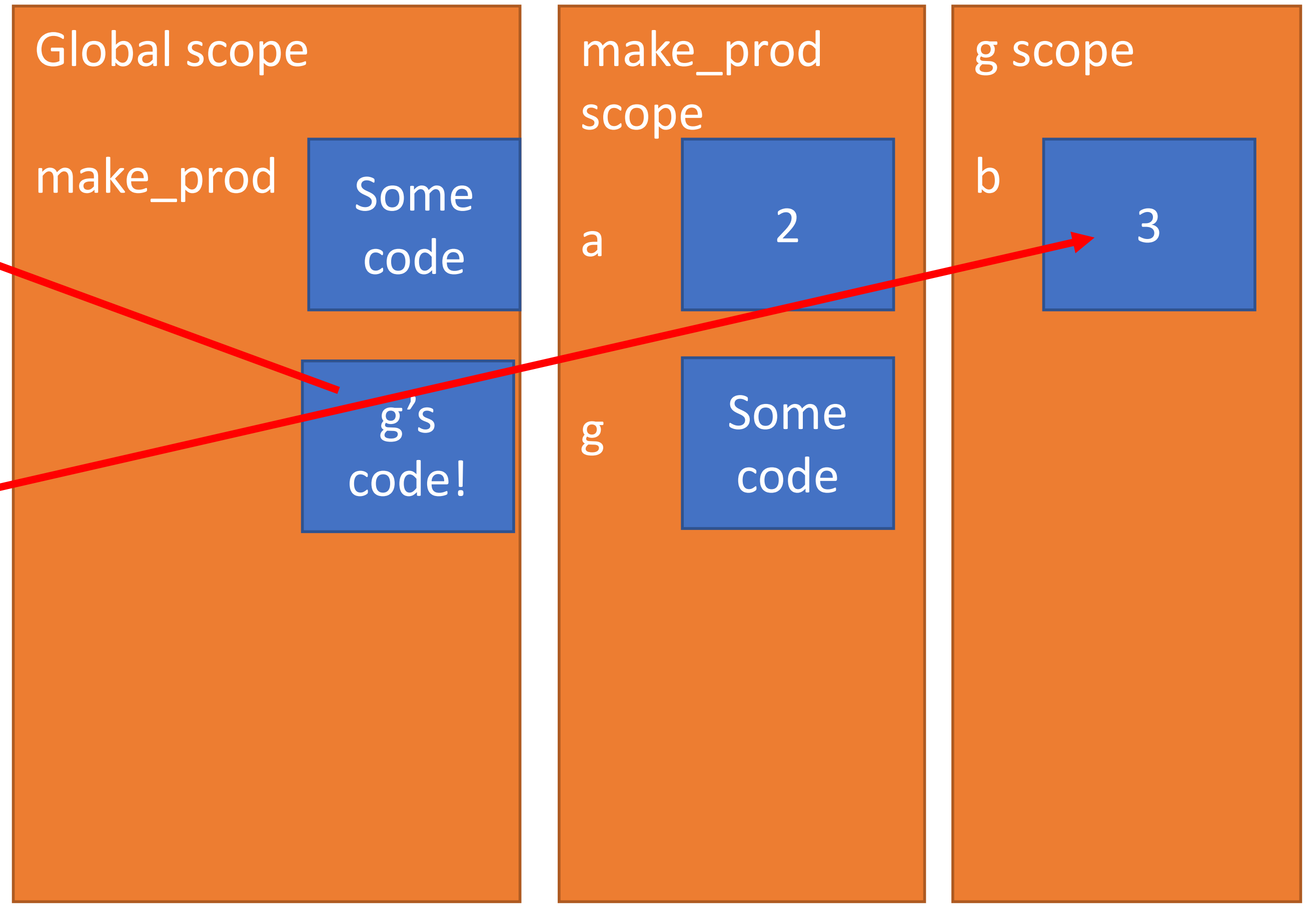
Evaluating `make_prod(2)` has returned an anonymous procedure

Returns pointer to g code

SCOPE DETAILS FOR WAY 1

```
def make_prod(a):
    def g(b):
        return a*b
    return g


val = make_prod(2)(3)

print(val)
```

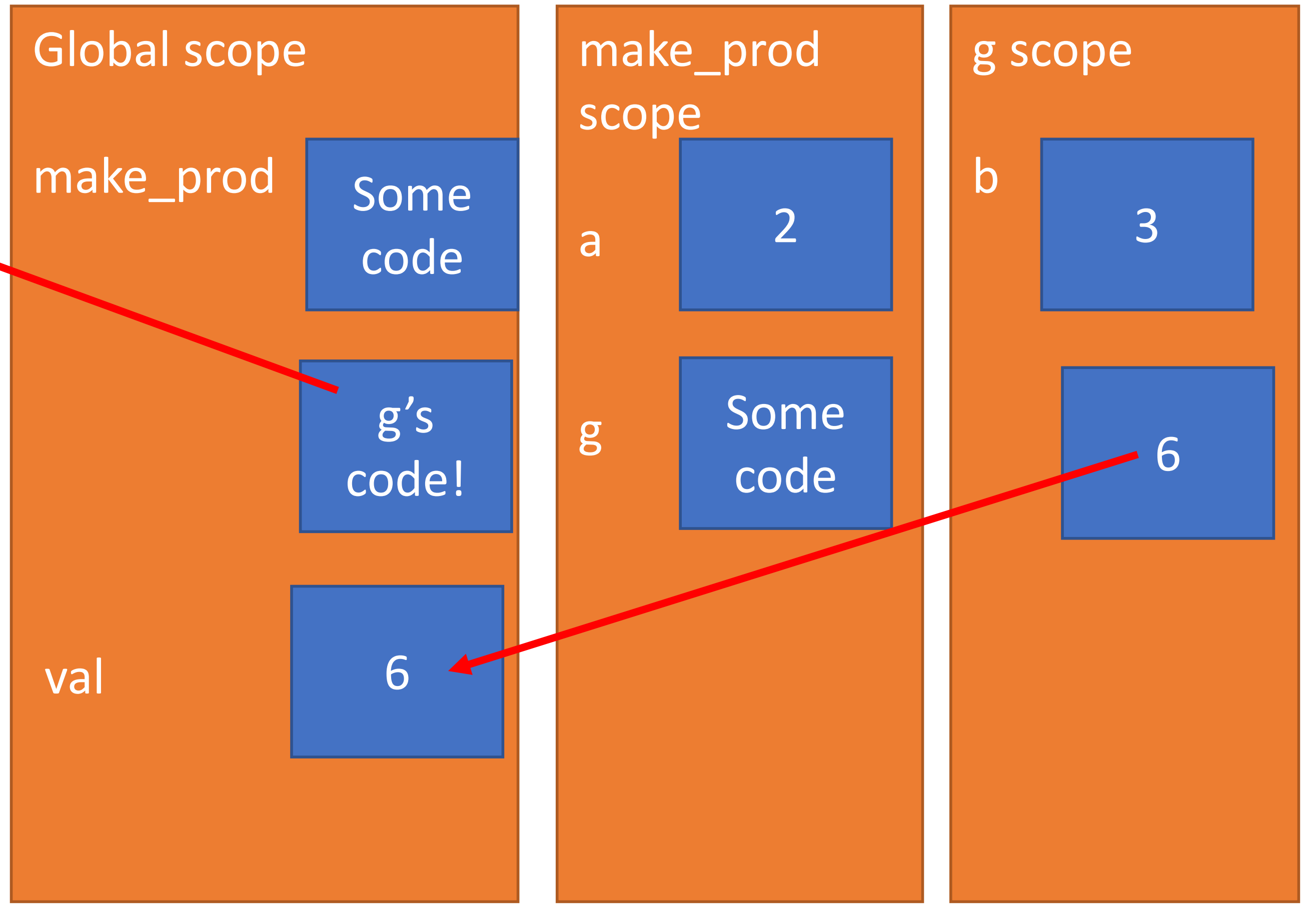
Call is g(3)



SCOPE DETAILS FOR WAY 1

```
def make_prod(a):
    def g(b):
        return a*b
    return g

val = make_prod(2)(3)
print(val)
```

Internal procedure only accessible within scope from parent procedure's call



How does g get value for a?
Interpreter can move up hierarchy of frames to see both b and a values

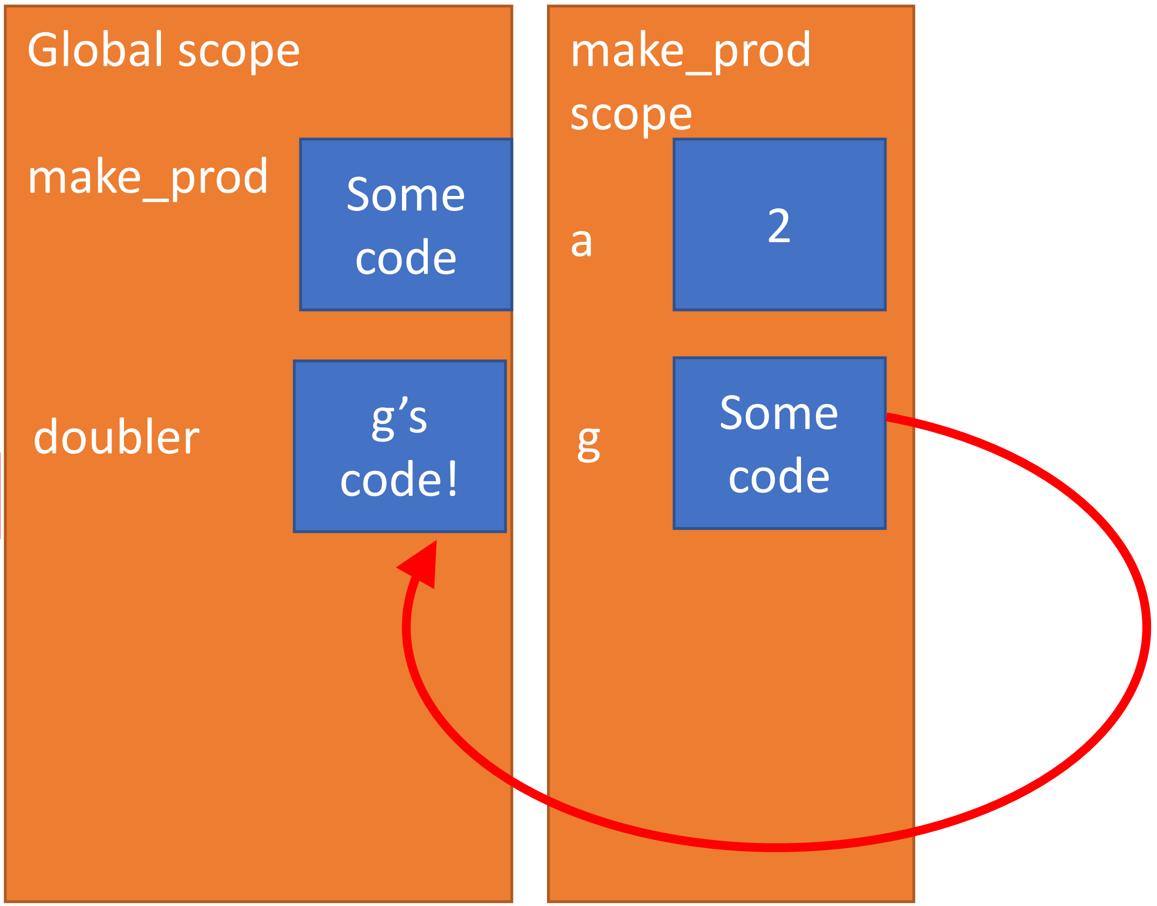
SCOPE DETAILS FOR WAY 2

```
def make_prod(a):
    def g(b):
        return a*b
    return g

doubler = make_prod(2)
val = doubler(3)
print(val)
```

SCOPE DETAILS FOR WAY 2

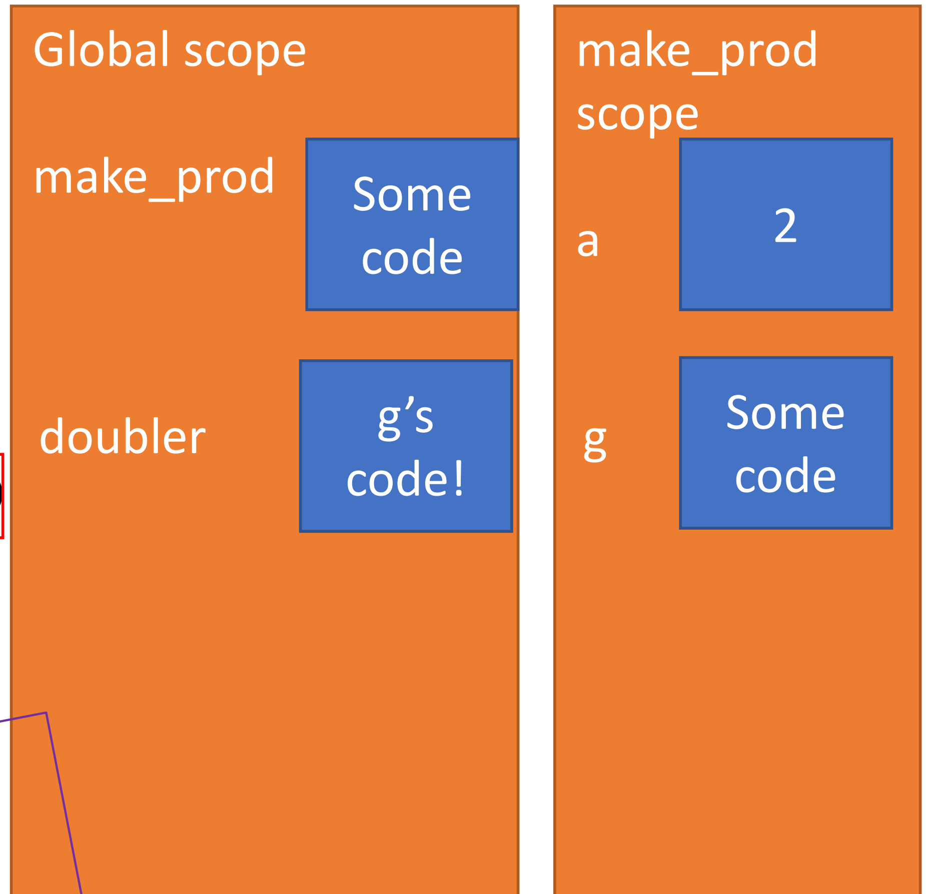
```
def make_prod(a):
    def g(b):
        return a*b
    return g

doubler = make_prod(2)
val = doubler(3)
print(val)
```

g



SCOPE DETAILS FOR WAY 2

```
def make_prod(a):
    def g(b):
        return a*b
    return g

doubler = make_prod(2)
val = doubler(3)
print(val)
```

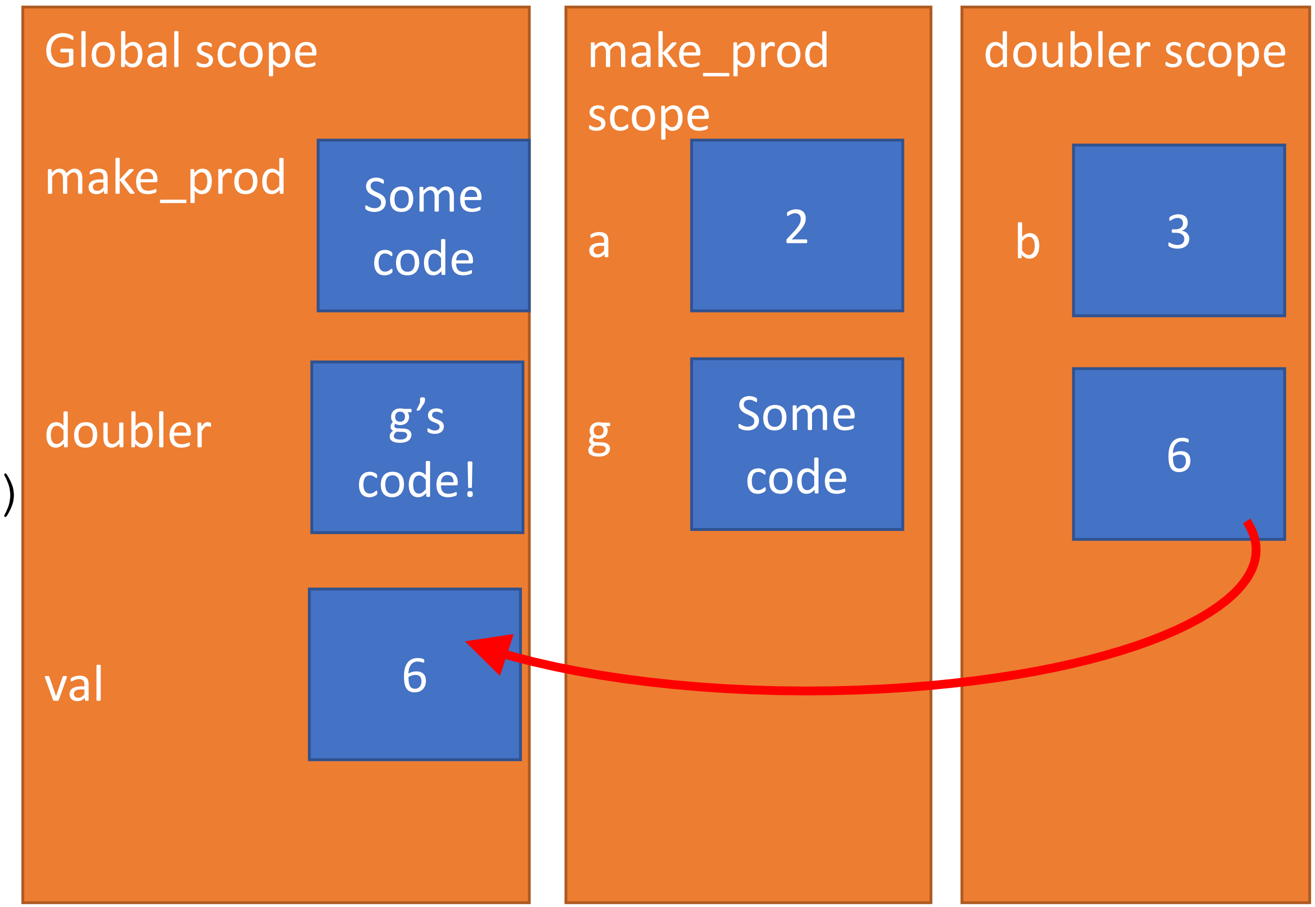


Evaluating `make_prod(2)` has same effect as previous example – doubler is a **function object**

SCOPE DETAILS FOR WAY 2

```
def make_prod(a):
    def g(b):
        return a*b
    return g

doubler = make_prod(2)
val = doubler(3)
print(val)
```

Now invoking g(3)



Returns value

WHY BOTHER RETURNING FUNCTIONS?

- Code can be **rewritten** without returning function objects
- Good software design
  - Embracing ideas of **decomposition**, **abstraction**
  - Another **tool** to structure code
- Interrupting execution
  - Example of **control flow**
  - A way to achieve **partial execution** and use result somewhere else before finishing the full evaluation

TESTING and DEBUGGING

DEFENSIVE PROGRAMMING

- Write **specifications** for functions
- **Modularize** programs
- Check **conditions** on inputs/outputs (assertions)

TESTING/VALIDATION

- **Compare** input/output pairs to specification
- "It's not working!"
- "How can I break my program?"

DEBUGGING

- **Study events** leading up to an error
- "Why is it not working?"
- "How can I fix my program?"

SET YOURSELF UP FOR EASY TESTING AND DEBUGGING

- From the **start**, design code to ease this part
- Break program up into **modules** that can be tested and debugged individually
- **Document constraints** on modules
  - What do you expect the input to be?
  - What do you expect the output to be?
- **Document assumptions** behind code design

WHEN ARE YOU READY TO TEST?

- Ensure **code runs**
  - Remove syntax errors
  - Remove static semantic errors
  - Python interpreter can usually find these for you
- Have a **set of expected results**
  - An input set
  - For each input, the expected output

CLASSES OF TESTS

- **Unit testing**
  - Validate each piece of program
  - **Testing each function** separately
- **Regression testing**
  - Add test for bugs as you find them
  - **Catch reintroduced** errors that were previously fixed
- **Integration testing**
  - Does **overall program** work?
  - Tend to rush to do this

TESTING APPROACHES

- **Intuition** about natural boundaries to the problem

```
def is_bigger(x, y):
    """ Assumes x and y are ints
    Returns True if y is less than x, else False """
```

  - can you come up with some natural partitions?
- If no natural partitions, might do **random testing**
  - Probability that code is correct increases with more tests
  - Better options below
- **Black box testing**
  - Explore paths through specification
- **Glass box testing**
  - Explore paths through code

BLACK BOX TESTING

```
def sqrt(x, eps):
    """ Assumes x, eps floats, x >= 0, eps > 0
    Returns res such that x-eps <= res*res <= x+eps """
```

- Designed **without looking** at the code
- Can be done by someone other than the implementer to avoid some implementer **biases**
- Testing can be **reused** if implementation changes
- **Paths** through specification
  - Build test cases in different natural space partitions
  - Also consider boundary conditions (empty lists, singleton list, large numbers, small numbers)

BLACK BOX TESTING

```
def sqrt(x, eps):
    """ Assumes x, eps floats, x >= 0, eps > 0
    Returns res such that x-eps <= res*res <= x+eps """
```

CASE	x	eps
boundary	0	0.0001
perfect square	25	0.0001
less than 1	0.05	0.0001
irrational square root	2	0.0001
extremes	2	1.0/2.0**64.0
extremes	1.0/2.0**64.0	1.0/2.0**64.0
extremes	2.0**64.0	1.0/2.0**64.0
extremes	1.0/2.0**64.0	2.0**64.0
extremes	2.0**64.0	2.0**64.0

GLASS BOX TESTING

- **Use code** directly to guide design of test cases
- Called **path-complete** if every potential path through code is tested at least once
- What are some **drawbacks** of this type of testing?
  - Can go through loops arbitrarily many times
  - Missing paths
- Guidelines
  - Branches
  - For loops
  - While loops

exercise all parts of a conditional

loop not entered
body of loop executed exactly once
body of loop executed more than once

same as for loops, cases that catch all ways to exit loop

GLASS BOX TESTING

```
def abs(x):
    """ Assumes x is an int
    Returns x if x>=0 and -x otherwise """
    if x < -1:
        return -x
    else:
        return x
```

- Aa path-complete test suite could **miss a bug**
- Path-complete test suite: 2 and -2
- But abs(-1) incorrectly returns -1
- Should still test boundary cases

DEBUGGING

- Once you have discovered that your code does not run properly, you want to:
  - Isolate the bug(s)
  - Eradicate the bug(s)
  - Retest until code runs correctly for all cases
  - Steep learning curve
- Goal is to have a bug-free program
- Tools
  - **Built in** to IDLE and Anaconda
  - **Python Tutor**
  - `print` statement
  - Use your brain, be **systematic** in your hunt

ERROR MESSAGES – EASY

- Trying to access beyond the limits of a list

  `test = [1,2,3]` then `test[4]` → `IndexError`
- Trying to convert an inappropriate type

  `int(test)` → `TypeError`
- Referencing a non-existent variable

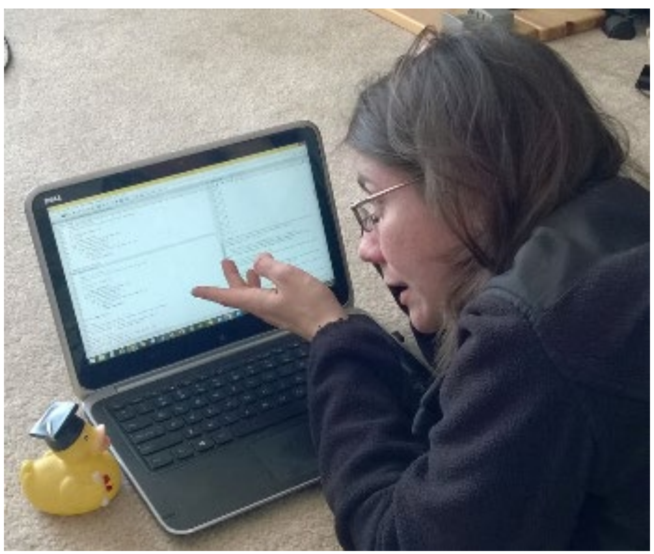
  `a` → `NameError`
- Mixing data types without appropriate coercion

  `'3'/4` → `TypeError`
- Forgetting to close parenthesis, quotation, etc.

```
a = len([1,2,3]
print(a)
```
→ `SyntaxError`

LOGIC ERRORS - HARD

- **think** before writing new code
- **draw** pictures, take a break
- **explain** the code to
  - someone else
  - a rubber ducky



DEBUGGING STEPS

- **Study** program code
  - Don’t ask what is wrong
  - Ask how did I get the unexpected result
  - Is it part of a family?
- **Scientific method**
  - Study available data
  - Form hypothesis
  - Repeatable experiments
  - Pick simplest input to test with

PRINT STATEMENTS

- Good way to **test hypothesis**
- When to print
  - Enter function
  - Parameters
  - Function results
- Use **bisection method**
  - Put print halfway in code
  - Decide where bug may be depending on values

EXCEPTIONS, ASSERTIONS

(download slides and .py files to follow along)

6.100L Lecture 13

Ana Bell

EXCEPTIONS

UNEXPECTED CONDITIONS

- What happens when procedure execution hits an **unexpected condition**?
- Get an **exception**… to what was expected
  - Trying to access beyond list limits

    `test = [1,7,4]`

    `test[4]` → `IndexError`
  - Trying to convert an inappropriate type

    `int(test)` → `TypeError`
  - Referencing a non-existing variable

    `a` → `NameError`
  - Mixing data types without coercion

    `'a'/4` → `TypeError`

HANDLING EXCEPTIONS

- Typically, exception causes an error to occur and execution to stop
- Python code can provide **handlers** for exceptions

```
try:
    # do some potentially
    # problematic code
except:
    # do something to
    # handle the problem
```

```
if <all potentially problematic code succeeds>:
    # great, all that code
    # just ran fine!
else:
    # do something to
    # handle the problem
```

- If expressions in **try block all succeed**
  - Evaluation continues with code after except block
- Exceptions **raised** by any statement in body of **`try`** are **handled** by the **`except`** statement
  - Execution continues with the body of the `except` statement
  - Then other expressions after that block of code

EXAMPLE with CODE YOU MIGHT HAVE ALREADY SEEN

- A function that sums digits in a string

CODE YOU'VE SEEN

```
def sum_digits(s):
  """ s is a non-empty string
      containing digits.
  Returns sum of all chars that
  are digits """
  total = 0
  for char in s:
    if char in '0123456789':
      val = int(char)
      total += val
  return total
```

Problematic if try to do
int('a')

CODE WITH EXCEPTIONS

```
def sum_digits(s):
  """ s is a non-empty string
      containing digits.
  Returns sum of all chars that
  are digits """
  total = 0
  for char in s:
    try:
      val = int(char)
      total += val
    except:
      print("can't convert", char)
  return total
```

Print and move on to next char

USER INPUT CAN LEAD TO EXCEPTIONS

- User might input a character :(
- User might make b be 0 :(

```
a = int(input("Tell me one number:"))
b = int(input("Tell me another number:"))
print(a/b)
```

- Use try/except around the problematic code

```
try:
    a = int(input("Tell me one number:"))
    b = int(input("Tell me another number:"))
    print(a/b)
except:
    print("Bug in user input.")
```

HANDLING SPECIFIC EXCEPTIONS

- Have **separate `except` clauses** to deal with a particular type of exception

```
try:
    a = int(input("Tell me one number: "))
    b = int(input("Tell me another number: "))
    print("a/b = ", a/b)
    print("a+b = ", a+b)
except ValueError:
    print("Could not convert to a number.")
except ZeroDivisionError:
    print("Can't divide by zero")
    print("a/b = infinity")
    print("a+b =", a+b)
except:
    print("Something went very wrong.")
```

only execute if these errors come up

for all other errors

OTHER BLOCKS ASSOCIATED WITH A TRY BLOCK

- `else`:
  - Body of this is executed when execution of associated `try` body **completes with no exceptions**
- `finally`:
  - Body of this is **always executed** after `try`, `else` and `except` clauses, even if they raised another error or executed a `break`, `continue` or `return`
  - Useful for clean-up code that should be run no matter what else happened (e.g. close a file)
- Nice to know these exist, but we don't really use these in this class

WHAT TO DO WITH EXCEPTIONS?

- What to do when encounter an error?
- **Fail silently**:
  - Substitute default values or just continue
  - Bad idea! user gets no warning
- Return an **"error" value**
  - What value to choose?
  - Complicates code having to check for a special value
- Stop execution, **signal error** condition
  - In Python: **raise an exception**

```
raise ValueError("something is wrong")
```

keyword

name of error you want to raise

optional, but typically a string with a message

EXAMPLE with SOMETHING YOU'VE ALREADY SEEN

- A function that sums digits in a string
- Execution stopping means a bad result is not propagated

```
def sum_digits(s):
    """ s is a non-empty string containing digits.
     Returns sum of all chars that are digits """
     total = 0
     for char in s:
         try:
             val = int(char)
             total += val
         except:
             raise ValueError("string contained a character")
     return total
```

Halt execution as soon as you see a non-digit with our own informative message. Does not go on to next char!

YOU TRY IT!

```
def pairwise_div(Lnum, Ldenom):
    """ Lnum and Ldenom are non-empty lists of equal lengths containing numbers

    Returns a new list whose elements are the pairwise
    division of an element in Lnum by an element in Ldenom.

    Raise a ValueError if Ldenom contains 0. """
    # your code here

# For example:
L1 = [4,5,6]
L2 = [1,2,3]
# print(pairwise_div(L1, L2))  # prints [4.0,2.5,2.0]

L1 = [4,5,6]
L2 = [1,0,3]
# print(pairwise_div(L1, L2))  # raises a ValueError
```

ASSERTIONS

ASSERTIONS: DEFENSIVE PROGRAMMING TOOL

- Want to be sure that **assumptions** on state of computation are as expected
- Use an **`assert` statement** to raise an `AssertionError` exception if assumptions not met

```
assert <statement that should be true>, "message if not true"
```

- An example of good **defensive programming**
  - Assertions don't allow a programmer to control response to unexpected conditions
  - Ensure that **execution halts** whenever an expected condition is not met
  - Typically used to **check inputs** to functions, but can be used anywhere
  - Can be used to **check outputs** of a function to avoid propagating bad values
  - Can make it easier to locate a source of a bug

EXAMPLE with SOMETHING YOU'VE ALREADY SEEN

- A function that sums digits in a **NON-EMPTY string**
- Execution stopping means a bad result is not propagated

```
def sum_digits(s):
    """ s is a non-empty string containing digits.
    Returns sum of all chars that are digits """
    assert len(s) != 0, "s is empty"
    total = 0
    for char in s:
        try:
            val = int(char)
            total += val
        except:
            raise ValueError("string contained a character")
```

Halt execution when specification is not met

YOU TRY IT!

```
def pairwise_div(Lnum, Ldenom):
    """ Lnum and Ldenom are non-empty lists of equal lengths
        containing numbers
    Returns a new list whose elements are the pairwise
    division of an element in Lnum by an element in Ldenom.
    Raise a ValueError if Ldenom contains 0. """
    # add an assert line here
```

ANOTHER EXAMPLE

LONGER EXAMPLE OF EXCEPTIONS and ASSERTIONS

- Assume we are **given a class list** for a subject: each entry is a list of two parts
  - A list of first and last name for a student
  - A list of grades on assignments

Two students, each with a name list and a grades list

```
test_grades = [[['peter', 'parker'], [80.0, 70.0, 85.0]],
               [['bruce', 'wayne'], [100.0, 80.0, 74.0]]]
```

- Create a **new class list**, with name, grades, and an average added at the end

```
[[['peter', 'parker'], [80.0, 70.0, 85.0], 78.33333],
 [['bruce', 'wayne'], [100.0, 80.0, 74.0], 84.666667]]]
```

EXAMPLE CODE

```
[[['peter', 'parker'], [80.0, 70.0, 85.0]],
 [['bruce', 'wayne'], [100.0, 80.0, 74.0]]]
```

elt is for example:
`[['peter', 'parker'], [80.0, 70.0, 85.0]]`

```
def get_stats(class_list):
    new_stats = []
    for stu in class_list:
        new_stats.append([stu[0], stu[1], avg(stu[1])])
    return new_stats

def avg(grades):
    return sum(grades)/len(grades)
```

We will look at variations of the avg function

ERROR IF NO GRADE FOR A STUDENT

- If one or more students **don't have any grades**, get an error

```
test_grades = [[['peter', 'parker'], [10.0,55.0,85.0]],
               [['bruce', 'wayne'], [10.0,80.0,75.0]],
               [['captain', 'america'], [80.0,10.0,96.0]],
               [['deadpool'], []]]
```

- Get `ZeroDivisionError: float division by zero` because try to `return sum(grades)/len(grades)`

length is 0

OPTION 1: FLAG THE ERROR BY PRINTING A MESSAGE

- Decide to **notify** that something went wrong with a msg

```
def avg(grades):
    try:
        return sum(grades)/len(grades)
    except ZeroDivisionError:
        print('warning: no grades data')
```

- Running on same test data gives

```
warning: no grades data
```

flagged the error

```
[[['peter', 'parker'], [10.0, 55.0, 85.0], 50.0],
[['bruce', 'wayne'], [10.0, 80.0, 75.0], 55.0],
[['captain', 'america'], [80.0, 10.0, 96.0], 62.0],
[['deadpool'], [], None]]
```

because avg did not return anything in the except

OPTION 2: CHANGE THE POLICY

- Decide that a student with no grades gets a **zero**

```
def avg(grades):
    try:
        return sum(grades)/len(grades)
    except ZeroDivisionError:
        print('warning: no grades data')
        return 0.0
```

- Running on same test data gives

```
warning: no grades data
[[['peter', 'parker'], [10.0, 55.0, 85.0], 50.0],
[['bruce', 'wayne'], [10.0, 80.0, 75.0], 55.0],
[['captain', 'america'], [80.0, 10.0, 96.0], 62]
[['deadpool'], [], 0.0]]
```

still flag the error

now avg returns 0

OPTION 3: HALT EXECUTION IF ASSERT IS NOT MET

function ends immediately if assertion not met

```
def avg(grades):
    assert len(grades) != 0, 'no grades data'
    return sum(grades)/len(grades)
```

- Raises an `AssertionError` if it is given an empty list for grades, prints out string message; stops execution
- Otherwise runs as normal

ASSERTIONS vs. EXCEPTIONS

- Goal is to **spot bugs as soon as introduced** and make clear where they happened
- Exceptions provide a way of **handling unexpected input**
  - Use when you don't need to halt program execution
  - Raise exceptions if users supplies bad data input
- Use **assertions:**
  - Enforce conditions on a "contract" between a coder and a user
  - As a **supplement** to testing
  - Check **types** of arguments or values
  - Check that **invariants** on data structures are met
  - Check **constraints** on return values
  - Check for **violations** of constraints on procedure (e.g. no duplicates in a list)

DICTIONARIES

(download slides and .py files to follow along)

6.100L Lecture 14

Ana Bell

HOW TO STORE STUDENT INFO

- Suppose we want to store and use grade information for a set of students
- Could store using separate lists for each kind of information

```
names  =   ['Ana',  'John',  'Matt',  'Katy']
grades =   ['A+' ,   'B'  ,    'A' ,    'A' ]
microquizzes = ...
psets = ...
```

- Info stored across lists at **same index**, each index refers to information for a different person
- Indirectly access information by finding location in lists corresponding to a person, then extract

HOW TO ACCESS STUDENT INFO

```
def get_grade(student, name_list, grade_list):
    i = name_list.index(student)
    grade = grade_list[i]
    return (student, grade)
```

find location in list for person

Use location index to access other info

- **Messy** if have a lot of different info of which to keep track, e.g., a separate list for microquiz scores, for pset scores, etc.
- Must maintain **many lists** and pass them as arguments
- Must **always index** using integers
- Must remember to change multiple lists, when adding or updating information

HOW TO STORE AND ACCESS STUDENT INFO

- Alternative might be to use a list of lists

```
eric = ['eric', ['ps', [8, 4, 5]], ['mq', [6, 7]]]
ana = ['ana', ['ps', [10, 10, 10]], ['mq', [9, 10]]]
john = ['john', ['ps', [7, 6, 5]], ['mq', [8, 5]]]

grades = [eric, ana, john]
```

- Then could access by searching lists, but code is still messy

```
def get_grades(who, what, data):
    for stud in data:
        if stud[0] == who:
            for info in stud[1:]:
                if info[0] == what:
                    return who, info

print(get_grades('eric', 'mq', grades))
print(get_grades('ana', 'ps', grades))
```

But idea of associating data with names is worth exploring

A BETTER AND CLEANER WAY – A DICTIONARY

- Nice to use **one data structure**, no separate lists
- Nice to **index item of interest directly**
- A Python **dictionary has entries** that map a key:value

A list

0	Elem 1
1	Elem 2
2	Elem 3
3	Elem 4
…	…

index element

A dictionary

Key 1	Val 1
Key 2	Val 2
Key 3	Val 3
Key 4	Val 4
…	…

custom index element

BIG IDEA

Dict value refers to the value associated with a key.

This terminology is may sometimes be confused with the regular value of some variable.

A PYTHON DICTIONARY

- Store **pairs of data** as an **entry**
  - key (any immutable object)
    - str, int, float, bool, tuple, etc
  - value (any data object)
    - Any above plus lists and other dicts!

'Ana'	'B'
'Matt'	'A'
'John'	'B'
'Katy'	'A'

custom index by label

element

empty dictionary

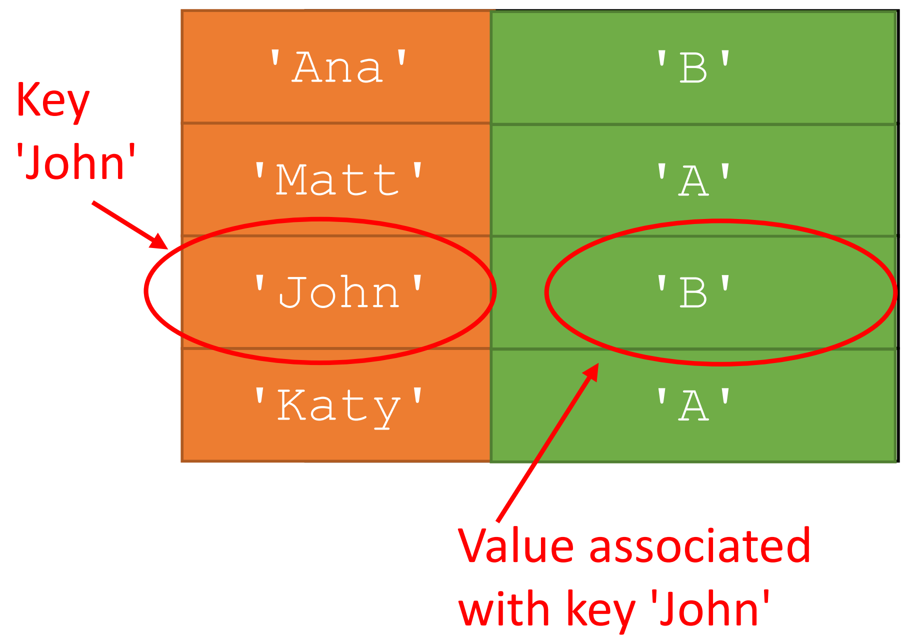
colon maps key:value

```
my_dict = {}
d = {4:16}
grades = {'Ana':'B', 'Matt':'A', 'John':'B', 'Katy':'A'}
```

key1 val1 key2 val2 key3 val3 key4 val4

DICTIONARY LOOKUP

- Similar to indexing into a list
- **Looks up** the **key**
- **Returns** the **value** associated with the key
  - If key isn’t found, get an error
- There is **no simple expression to get a key back given some value**!




```
grades = {'Ana':'B', 'Matt':'A', 'John':'B', 'Katy':'A'}
```

`grades['John']` → evaluates to `'B'`

`grades['Grace']` → gives a `KeyError`

YOU TRY IT!

- Write a function according to this spec

```
def find_grades(grades, students):
    """ grades is a dict mapping student names (str) to grades (str)
        students is a list of student names
    Returns a list containing the grades for students (in same order) """

# for example

d = {'Ana':'B', 'Matt':'C', 'John':'B', 'Katy':'A'}
print(find_grades(d, ['Matt', 'Katy'])) # returns ['C', 'A']
```

BIG IDEA

Getting a dict value is just a matter of indexing with a key.

No. Need. To. Loop

DICTIONARY OPERATIONS

'Ana'	'B'
'Matt'	'A'
'John'	'B'
'Katy'	'A'
'Grace'	'C'

```
grades = {'Ana':'B', 'Matt':'A', 'John':'B', 'Katy':'A'}
```

- **Add** an entry

```
grades['Grace'] = 'A'
```

An assignment statement, but to a location in a dictionary – different from a list

- **Change** entry

```
grades['Grace'] = 'C'
```

- **Delete** entry

```
del(grades['Ana'])
```

Note that the dictionary is being mutated!

DICTIONARY OPERATIONS

'Ana'	'B'
'Matt'	'A'
'John'	'B'
'Katy'	'A'

```
grades = {'Ana':'B', 'Matt':'A', 'John':'B', 'Katy':'A'}
```

- **Test** if key in dictionary

```
'John' in grades          → returns True
'Daniel' in grades        → returns False
'B' in grades             → returns False
```

The in keyword only checks keys, not values

YOU TRY IT!

- Write a function according to these specs

```
def find_in_L(Ld, k):
    """ Ld is a list of dicts
        k is an int
    Returns True if k is a key in any dicts of Ld and False otherwise """

# for example
d1 = {1:2, 3:4, 5:6}
d2 = {2:4, 4:6}
d3 = {1:1, 3:9, 4:16, 5:25}

print(find_in_L([d1, d2, d3], 2)  # returns True
print(find_in_L([d1, d2, d3], 25)  # returns False
```

DICTIONARY OPERATIONS

'Ana'	'B'
'Matt'	'A'
'John'	'B'
'Katy'	'A'

- Can iterate over dictionaries but assume there is no guaranteed order

```
grades = {'Ana':'B', 'Matt':'A', 'John':'B', 'Katy':'A'}
```

- Get an **iterable that acts like a tuple of all keys**

`grades.keys()` → returns `dict_keys(['Ana', 'Matt', 'John', 'Katy'])`

`list(grades.keys())` → returns `['Ana', 'Matt', 'John', 'Katy']`

- Get an **iterable that acts like a tuple of all dict values**

`grades.values()` → returns `dict_values(['B', 'A', 'B', 'A'])`

`list(grades.values())` → returns `['B', 'A', 'B', 'A']`

DICTIONARY OPERATIONS
most useful way to iterate over dict entries (both keys and vals!)

'Ana'	'B'
'Matt'	'A'
'John'	'B'
'Katy'	'A'

- Can iterate over dictionaries but assume there is no guaranteed order

```
grades = {'Ana':'B', 'Matt':'A', 'John':'B', 'Katy':'A'}
```

- Get an **iterable that acts like a tuple of all items**

`grades.items()`

→ returns `dict_items([('Ana', 'B'), ('Matt', 'A'), ('John', 'B'), ('Katy', 'A')])`

`list(grades.items())`

→ returns `[('Ana', 'B'), ('Matt', 'A'), ('John', 'B'), ('Katy', 'A')]`

- Typical use is to **iterate over key,value tuple**

```
for k,v in grades.items():
    print(f"key {k} has value {v}")
```

key Ana has value B
key Matt has value A
key John has value B
key Katy has value A

YOU TRY IT!

- Write a function that meets this spec

```
def count_matches(d):
    """ d is a dict
    Returns how many entries in d have the key equal to its value """

# for example
d = {1:2, 3:4, 5:6}
print(count_matches(d))   # prints 0
d = {1:2, 'a':'a', 5:5}
print(count_matches(d))   # prints 2
```

DICTIONARY KEYS & VALUES

- Dictionaries are **mutable** objects (aliasing/cloning rules apply)
  - Use = sign to make an alias
  - Use d.copy() to make a copy
- **Assume there is no order** to keys or values!
- Dict values
  - Any type (**immutable and mutable**)
    - Dictionary values can be lists, even other dictionaries!
  - Can be **duplicates**
- Keys
  - Must be **unique**
  - **Immutable** type (`int`, `float`, `string`, `tuple`, `bool`)
    - Actually need an object that is **hashable,** but think of as immutable as all immutable types are hashable
  - Be careful using `float` type as a key

WHY IMMUTABLE/HASHABLE KEYS?

- A dictionary is stored in memory in a special way
- Next slides show an example

- Step 1: A **function is run on the dict key**
  - The function **maps any object to an int**
    E.g. map “a” to 1, “b” to 2, etc, so “ab” could map to 3
  - The int corresponds to a position in a block of memory addresses
- Step 2: At that memory address, **store the dict value**
- To do a **lookup** using a key, **run the same function**
  - If the object is immutable/hashable then you get the same int back
  - If the object is changed then the function gives back a different int!

Hash function:

1) Sum the letters
2) Take mod 16 (to fit in a memory block with 16 entries)

1 + 14 + 1 = 16
16%16 = 0

A n a	C

5 + 18 + 9 + 3 = 35
35%16 = 3

E r i c	A

10 + 15 + 8 + 14 = 47
47%16 = 15

J o h n	B

11 + 1 + 20 + 5 = 37
37%16 = 5

[K, a, t, e]	B

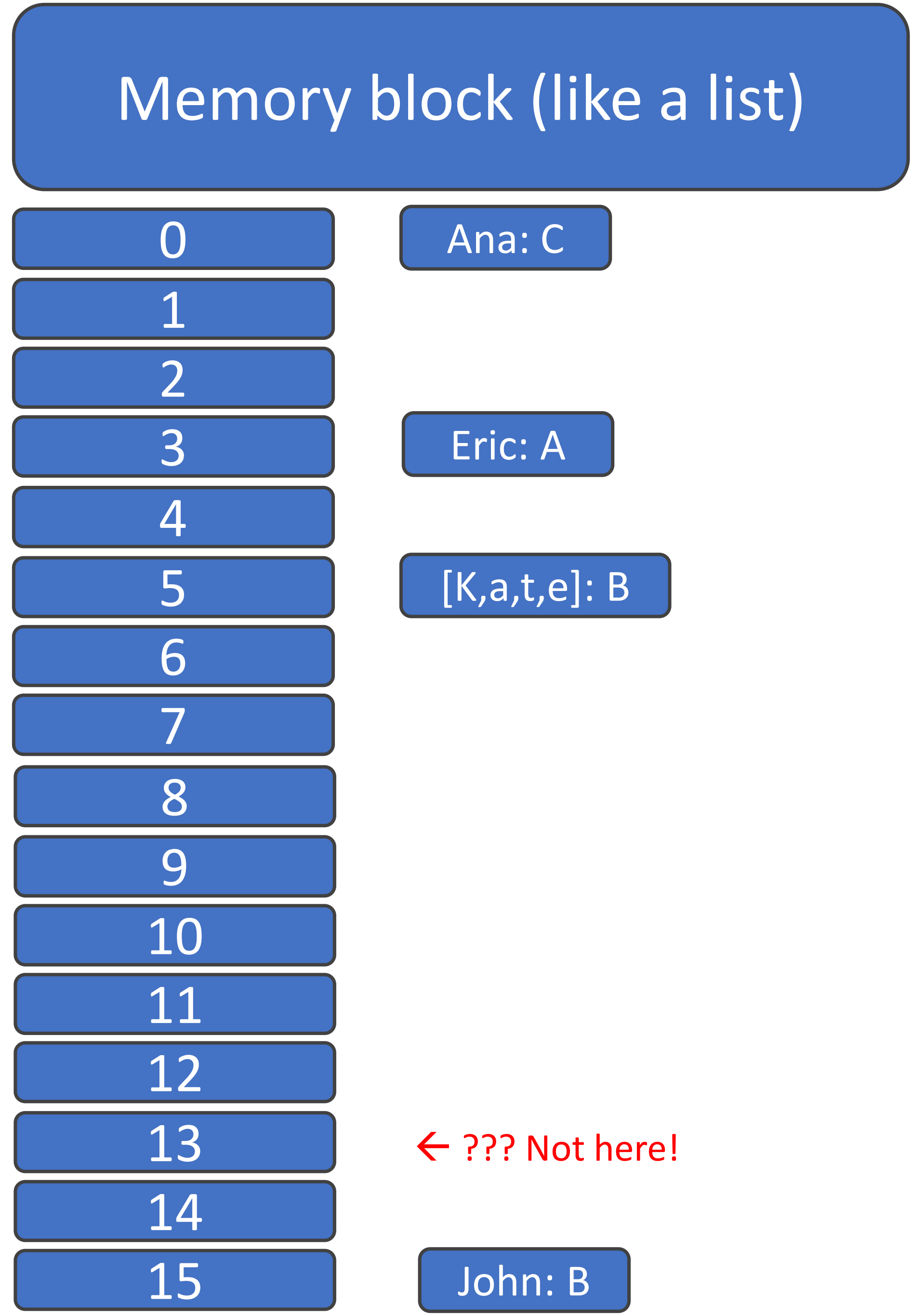
Memory block (like a list)

0	Ana: C
1	
2	
3	Eric: A
4	
5	[K,a,t,e]: B
6	
7	
8	
9	
10	
11	
12	
13	
14	
15	John: B

Hash function:

1) Sum the letters
2) Take mod 16 (to fit in a memory block with 16 entries)

Kate changes her name to Cate. Same person, different name. Look up her grade?



3 + 1 + 20 + 5 = 29

29%16 = 13

[C, a, t, e]

A PYTHON DICTIONARY for STUDENT GRADES

- Separate students are separate dict entries
- Entries are separated using a comma


```
grades = {'Ana':{'mq':[5,4,4], 'ps': [10,9,9], 'fin': 'B'},
          'Bob':{'mq':[6,7,8], 'ps': [8,9,10], 'fin': 'A'}}
```

A PYTHON DICTIONARY for STUDENT GRADES

- Each dict entry maps a key to a value
- The mapping is done with a : character
- grades maps str:dict

str	dict	
'Ana'	'mq'	[5,4,4]
	'ps'	[10,9,9]
	'fin'	'B'
'Bob'	'mq'	[6,7,8]
	'ps'	[8,9,10]
	'fin'	'A'

```
grades = {'Ana':{'mq':[5,4,4], 'ps': [10,9,9], 'fin': 'B'},
          'Bob':{'mq':[6,7,8], 'ps': [8,9,10], 'fin': 'A'}}
```

A PYTHON DICTIONARY for STUDENT GRADES

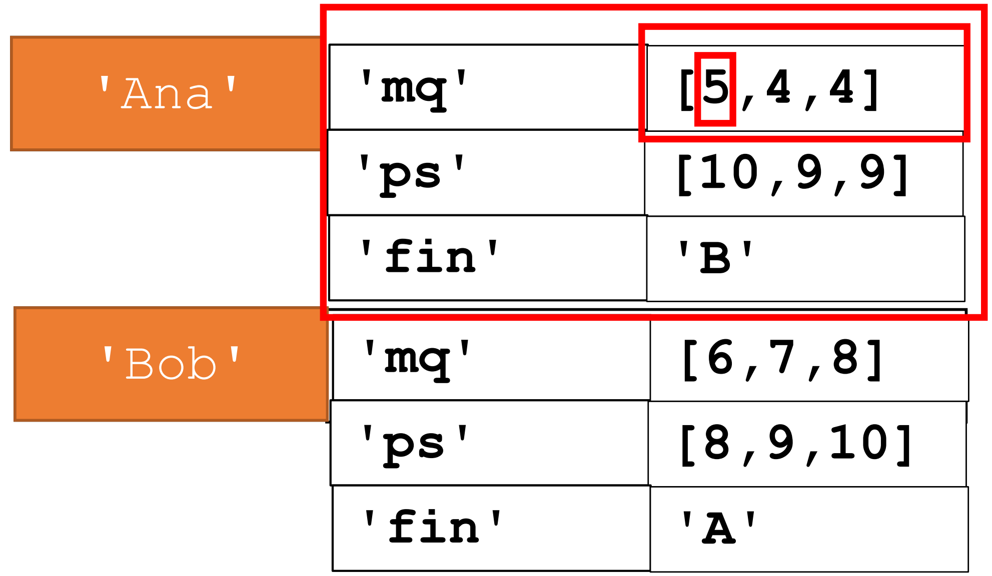
- The values of grades are dicts
- Each value maps a
  - str:list
  - str:str

'Ana'	'mq'	[5,4,4]
	'ps'	[10,9,9]
	'fin'	'B'
'Bob'	'mq'	[6,7,8]
	'ps'	[8,9,10]
	'fin'	'A'

```
grades = {'Ana':{'mq':[5,4,4], 'ps': [10,9,9], 'fin': 'B'},
          'Bob':{'mq':[6,7,8], 'ps': [8,9,10], 'fin': 'A'}}
```

A PYTHON DICTIONARY for STUDENT GRADES

- The values of grades are dicts
- Each value maps a
  - str:list
  - str:str




```
grades = {'Ana':{'mq':[5,4,4], 'ps': [10,9,9], 'fin': 'B'},
          'Bob':{'mq':[6,7,8], 'ps': [8,9,10], 'fin': 'A'}}
```

`grades['Ana']['mq'][0]` **returns** `5`

YOU TRY IT!

```
my_d ={'Ana':{'mq':[10], 'ps':[10,10]},
       'Bob':{'ps':[7,8], 'mq':[8]},
       'Eric':{'mq':[3], 'ps':[0]}       }

def get_average(data, what):
    all_data = []
    for stud in data.keys():
        INSERT LINE HERE
    return sum(all_data)/len(all_data)
```

Given the dict `my_d`, and the outline of a function to compute an average, which line should be inserted where indicated so that `get_average(my_d, 'mq')` computes average for all 'mq' entries? i.e. find average of all mq scores for all students.

A) `all_data = all_data + data[stud][what]`
B) `all_data.append(data[stud][what])`
C) `all_data = all_data + data[stud[what]]`
D) `all_data.append(data[stud[what]])`

`list` vs `dict`

- **Ordered** sequence of elements
- Look up elements by an integer index
- Indices have an **order**
- Index is an **integer**
- Value can be any type

- **Matches** “keys” to “values”
- Look up one item by another item
- **No order** is guaranteed
- Key can be any **immutable** type
- Value can be any type

EXAMPLE: FIND MOST COMMON WORDS IN A SONG'S LYRICS

1) Create a **frequency dictionary** mapping `str:int`

2) Find **word that occurs most often** and how many times

- Use a list, in case more than one word with same number
- Return a tuple `(list,int)` for (words_list, highest_freq)

3) Find the **words that occur at least X times**

- Let user choose "at least X times", so allow as parameter
- Return a list of tuples, each tuple is a `(list, int)` containing the list of words ordered by their frequency
- IDEA: From song dictionary, find most frequent word. Delete most common word. Repeat. It works because you are mutating the song dictionary.

CREATING A DICTIONARY
Python Tutor LINK

```
song = "RAH RAH AH AH AH ROM MAH RO MAH MAH"
def generate_word_dict(song):
    song_words = song.lower()
    words_list = song_words.split()
    word_dict = {}
    for w in words_list:
        if w in word_dict:
            word_dict[w] += 1
        else:
            word_dict[w] = 1
    return word_dict
```

Convert all chars to lower case

Convert string to list of words; divides based on spaces

Can iterate over list of words in song

If word in dict (as a key), increase # times you've seen it, update entry

If word not in dict, first time seeing word, create entry

Return is a dict mapping str:int

USING THE DICTIONARY

Python Tutor LINK

```
word_dict = {'rah':2, 'ah':3, 'rom':1, 'mah':3, 'ro':1}

def find_frequent_word(word_dict):
    words = []
    highest = max(word_dict.values())
    for k,v in word_dict.items():
        if v == highest:
            words.append(k)
    return (words, highest)
```

Highest frequency in dict's values

Loop to see which word has the highest freq

Append to list of all words that have that highest freq

Return is a tuple of (['ah', 'mah'], 3)

FIND WORDS WITH FREQUENCY GREATER THAN x=1

- Repeat the next few steps as long as the highest frequency is greater than x
- Find highest frequency

```
word_dict = {'rah':2, 'ah':3, 'rom':1, 'mah':3, 'ro':1}
```

FIND WORDS WITH FREQUENCY GREATER THAN x=1

- Use function `find_frequent_word` to get words with the biggest frequency

```
word_dict = {'rah':2, 'ah':3, 'rom':1, 'mah':3, 'ro':1}
```

FIND WORDS WITH FREQUENCY GREATER THAN x=1

- Remove the entries corresponding to these words from dictionary by mutation

```
word_dict = {'rah':2,          'rom':1,          'ro':1}
```

- Save them in the result

```
freq_list = [(['ah','mah'],3)]
```

FIND WORDS WITH FREQUENCY GREATER THAN x=1

- Find highest frequency in the mutated dict

```
word_dict = {'rah':2,          'rom':1,          'ro':1}
```

- The result so far…

```
freq_list = [(['ah','mah'],3)]
```

FIND WORDS WITH FREQUENCY GREATER THAN x=1

- Use function `find_frequent_word` to get words with that frequency

```
word_dict = {'rah':2,          'rom':1,          'ro':1}
```

- The result so far…

```
freq_list = [(['ah','mah'],3)]
```

FIND WORDS WITH FREQUENCY GREATER THAN x=1

- Remove the entries corresponding to these words from dictionary by mutation

```
word_dict = {                    'rom':1,             'ro':1}
```

- Add them to the result so far

```
freq_list = [(['ah','mah'],3), $\boxed{(['rah'],2)}$]
```

FIND WORDS WITH FREQUENCY GREATER THAN x=1

- The highest frequency is now smaller than x=2, so stop

```
word_dict = {                    'rom':1,          'ro':1}
```

- The final result

```
freq_list = [(['ah','mah'],3), (['rah'],2)]
```

LEVERAGING DICT PROPERTIES
Python Tutor LINK

```
word_dict = {'rah':2, 'ah':3, 'rom':1, 'mah':3, 'ro':1}

def occurs_often(word_dict, x):
    freq_list = []
    word_freq_tuple = find_frequent_word(word_dict)

    while word_freq_tuple[1] > x:

        word_freq_tuple = find_frequent_word(word_dict)
        freq_list.append(word_freq_tuple)
        for word in word_freq_tuple[0]:
            del(word_dict[word])
    return freq_list
```

Gives us a word tuple
Like (['ah', 'mah'], 3)

Stay in loop while we still have
frequencies higher than x

Add those words to result

Mutate dict to remove ALL
those words; on next loop, will
find next most common words

SOME OBSERVATIONS

- Conversion of **string into list** of words enables use of list methods
  - Used `words_list = song_words.split()`
- **Iteration over list** naturally follows from structure of lists
  - Used `for w in words_list:`
- Dictionary stored the **same data in a more appropriate way**
- Ability to **access all values and all keys** of dictionary allows natural looping methods
  - Used `for k,v in word_dict.items():`
- **Mutability of dictionary** enables iterative processing
  - Used `del(word_dict[word])`
- **Reused functions** we already wrote!

SUMMARY

- Dictionaries have entries that **map a key to a value**
- **Keys are immutable/hashable and unique** objects
- **Values** can be **any object**
- Dictionaries can make code efficient
    - Implementation-wise
    - Runtime-wise

6.100L Introduction to Computer Science and Programming Using Python
Fall 2022

RECURSION

(download slides and .py files to follow along)

6.100L Lecture 15

Ana Bell

ITERATIVE ALGORITHMS SO FAR

- Looping constructs (`while` and `for` loops) lead to **iterative** algorithms
- Can capture computation in a set of **state variables** that update, based on a set of rules, on each iteration through loop
  - What is **changing each time** through loop, and how?
  - How do I **keep track** of number of times through loop?
  - When can I **stop**?
  - Where is the **result** when I stop?

MULTIPLICATION

- The * operator does this for us
- Make a function

```
def mult(a, b):
    return a*b
```

MULTIPLICATION THINK in TERMS of ITERATION

- Can you make this iterative?
- Define `a*b` as `a+a+a+a... b` times
- Write a function

```
def mult(a, b):
    total = 0
    for n in range(b):
        total += a
    return total
```

MULTIPLICATION –
ANOTHER ITERATIVE SOLUTION

- "multiply `a` * `b`" is equivalent to "add `b` copies of `a`"

```
a + a + a + a + … + a
```

- Capture **state** by

Update rules

  - An **iteration** number (`i`) starts at b
    `i` ← `i-1` and stop when 0
  - A current **value of computation** (`result`) starts at 0
    `result` ← `result + a`

```
def mult_iter(a, b):
    result = 0
    while b > 0:
        result += a
        b -= 1
    return result
```

MULTIPLICATION
NOTICE the RECURSIVE PATTERNS

- Recognize that we have a problem we are solving many times
- If **a = 5** and **b = 4**
  - 5*4 is 5+5+5+5
- **Decompose** the original problem into
  - **Something you know** and
  - the **same problem** again
- Original problem is using * between two numbers
  - $\boxed{5*4}$ is 5+$\boxed{5*\mathbf{3}}$
  - But this is 5+5+$\boxed{5*\mathbf{2}}$
  - And this is 5+5+5+$\boxed{5*\mathbf{1}}$

Original problem

A very similar problem with one small change

MULTIPLICATION
FIND SMALLER VERSIONS of the PROBLEM

- Recognize that we have a problem we are solving many times
- If **a = 5** and **b = 4**
  - 5*4 is 5+5+5+5
- **Decompose** the original problem into
  - **Something you know** and
  - the **same problem** again
- Original problem is using * between two numbers
  - $\boxed{\texttt{5*4}}$
  - $=\ \boxed{\texttt{5+(5*3)}}$
  - `= 5+(5+( 5*2 ))`
  - `= 5+(5+(5+(5*1)))`

Original problem

A multiplication with 5 is 5+5*one_less

MULTIPLICATION
FIND SMALLER VERSIONS of the PROBLEM

- Recognize that we have a problem we are solving many times
- If **a = 5** and **b = 4**
  - 5*4 is 5+5+5+5
- **Decompose** the original problem into
  - **Something you know** and
  - the **same problem** again
- Original problem is using * between two numbers
  - `5*4`
  - `= 5+(` $\boxed{\texttt{5*3}}$ `)`
  - `= 5+(` $\boxed{\texttt{5+(5*2)}}$ `)`
  - `= 5+(5+(5+(5*1)))`

Similar problem

A multiplication with 5 is 5+5*one_less

MULTIPLICATION
FIND SMALLER VERSIONS of the PROBLEM

- Recognize that we have a problem we are solving many times
- If **a = 5** and **b = 4**
  - 5*4 is 5+5+5+5
- **Decompose** the original problem into
  - **Something you know** and
  - the **same problem** again
- Original problem is using * between two numbers
  - `5*4`
  - `= 5+( 5*3 )`
  - `= 5+(5+( 5*2 ))`
  - `= 5+(5+(5+(5*1)))`

Similar problem

A multiplication with 5 is 5+5*one_less

MULTIPLICATION REACHED the END

- Recognize that we have a problem we are solving many times
- If **a = 5** and **b = 4**
  - 5*4 is 5+5+5+5
- **Decompose** the original problem into
  - **Something you know** and
  - the **same problem** again
- Original problem is using * between two numbers
  - `5*4`
  - `= 5+(      5*3      )`
  - `= 5+(5+(   5*2   ))`
  - `= 5+(5+(5+(5*1)))`

Basic fact: a number multiplied with itself is the same number.

MULTIPLICATION
BUILD the RESULT BACK UP

- Recognize that we have a problem we are solving many times
- If **a = 5** and **b = 4**
  - 5*4 is 5+5+5+5
- **Decompose** the original problem into
  - **Something you know** and
  - the **same problem** again
- Original problem is using * between two numbers
  - `5*4`
  - `= 5+( 5*3 )`
  - `= 5+(5+( 5*2 ))`
  - `= 5+(5+(5+( 5 )))`

Similar problem

10

MULTIPLICATION
BUILD the RESULT BACK UP

- Recognize that we have a problem we are solving many times
- If **a = 5** and **b = 4**
  - 5*4 is 5+5+5+5
- **Decompose** the original problem into
  - **Something you know** and
  - the **same problem** again
- Original problem is using * between two numbers
  - `5*4`
  - `= 5+(` $\boxed{\texttt{5*3}}$ `)`
  - `= 5+(` $\boxed{\texttt{5+(10)}}$ `)`
  - `= 5+(5+(5+( 5 )))`

Similar problem

15

MULTIPLICATION
BUILD the RESULT BACK UP

- Recognize that we have a problem we are solving many times
- If **a = 5** and **b = 4**
  - 5*4 is 5+5+5+5
- **Decompose** the original problem into
  - **Something you know** and
  - the **same problem** again
- Original problem is using * between two numbers
  - `5*4`
  - `= 5+(     15      )`
  - `= 5+(5+(   10    ))`
  - `= 5+(5+(5+(  5  )))`

Original problem

20

MULTIPLICATION – RECURSIVE and BASE STEPS

- **Recursive step**
  - Decide how to reduce problem to a **simpler/smaller version** of same problem, plus simple operations

$$\boxed{\texttt{a*b}} = \underbrace{\texttt{a + a + a + a + ... + a}}_{\text{b times}}$$

$$= \texttt{a + } \underbrace{\texttt{a + a + a + ... + a}}_{\text{b-1 times}}$$

$$= \texttt{a + } \boxed{\texttt{a * (b-1)}}$$

recursive reduction

MULTIPLICATION – RECURSIVE and BASE STEPS

- **Recursive step**
  - Decide how to reduce problem to a **simpler/smaller version** of same problem, plus simple operations
- **Base case**
  - Keep reducing problem until reach a simple case that can be **solved directly**
  - When `b=1, a*b=a`

$$\boxed{a*b} = \underbrace{a + a + a + a + \ldots + a}_{\text{b times}}$$

$$= a + \underbrace{a + a + a + a + \ldots + a}_{\text{b-1 times}}$$

$$= a + \boxed{a * (b-1)}$$

recursive reduction

MULTIPLICATION – RECURSIVE CODE Python Tutor LINK

- **Recursive step**
  - `If b != 1, a*b = a + a*(b-1)`
- **Base case**
  - `If b = 1, a*b = a`

```
def mult_recur(a, b):
    if b == 1:
        return a
    else:
        return a + mult_recur(a, b-1)
```

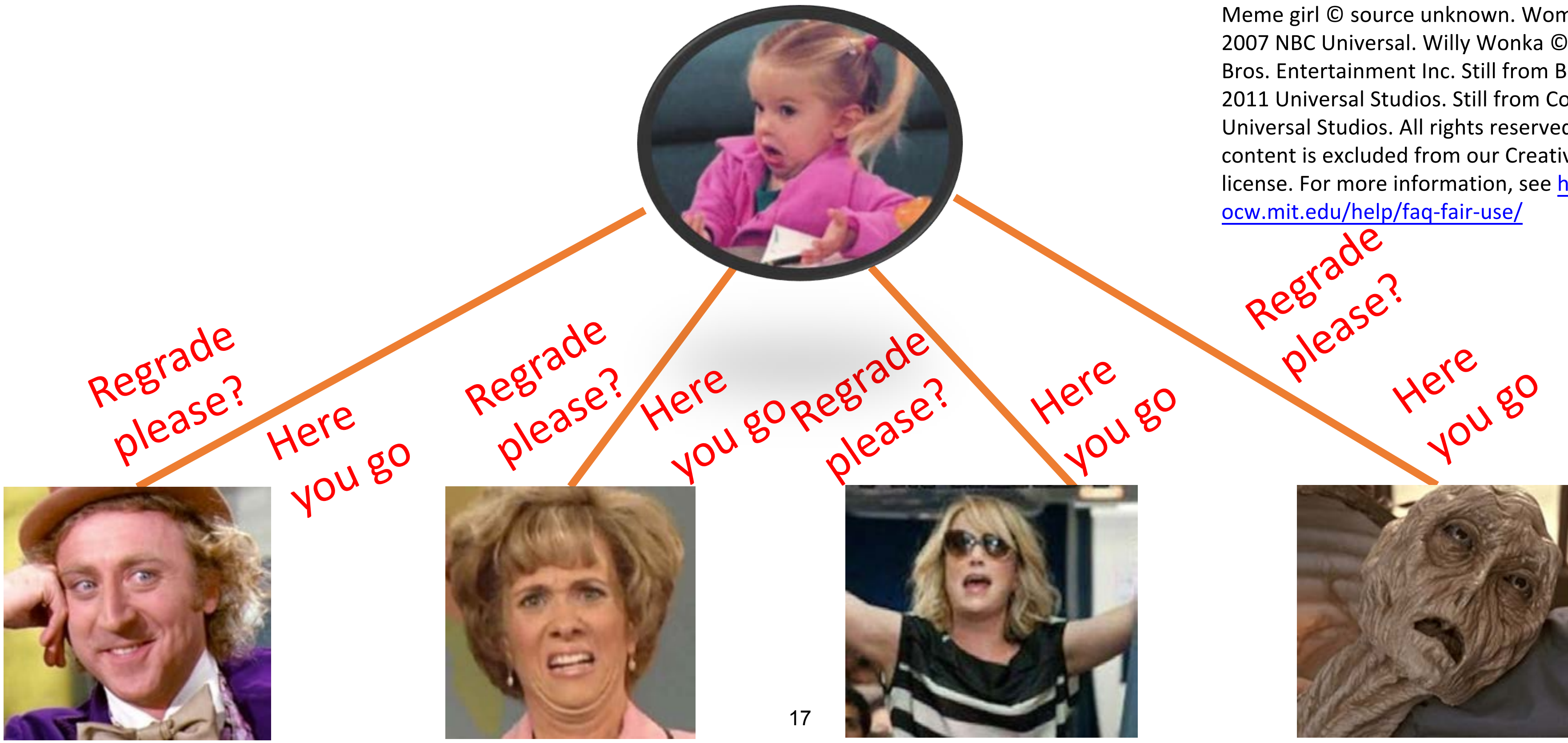
base case

recursive step

REAL LIFE EXAMPLE
Student requests a regrade: **ONLY ONE function call**

Iterative:

- Student asks the prof then the TA then the LA then the grader **one-by-one until one or more regrade the exam/parts**
- Student iterates through everyone and keeps track of the new score



REAL LIFE EXAMPLE
Student requests a regrade: **MANY function calls**

Recursive:

- 1) Student request(a **function call** to regrade!):
  - Asks the prof to regrade
  - Prof asks a TA to regrade
  - TA asks an LA to regrade
  - LA asks a grader to regrade

- 2) Relay the results (**functions return results** to their callers):
  - Grader tells the grade to the LA
  - LA tells the grade to the TA
  - TA tells the grade to the prof
  - Prof tells the grade to the student

BIG IDEA

"Earlier" function calls are waiting on results before completing.

WHAT IS RECURSION?

- Algorithmically: a way to design solutions to problems by **divide-and-conquer** or **decrease-and-conquer**
  - Reduce a problem to simpler versions of the same problem or to problem that can be solved directly
- Semantically: a programming technique where a **function calls itself**
  - In programming, goal is to NOT have infinite recursion
  - Must have **1 or more base cases** that are easy to solve directly
  - Must solve the same problem on **some other input** with the goal of simplifying the larger input problem, ending at base case

YOU TRY IT!

- Complete the function that calculates n^p for variables n and p

```
def power_recur(n, p):
    if ________:
        return ______
    elif ________:
        return ______
    else:
        return _________________
```

FACTORIAL

`n! = n*(n-1)*(n-2)*(n-3)* … * 1`

- For what `n` do we know the factorial?

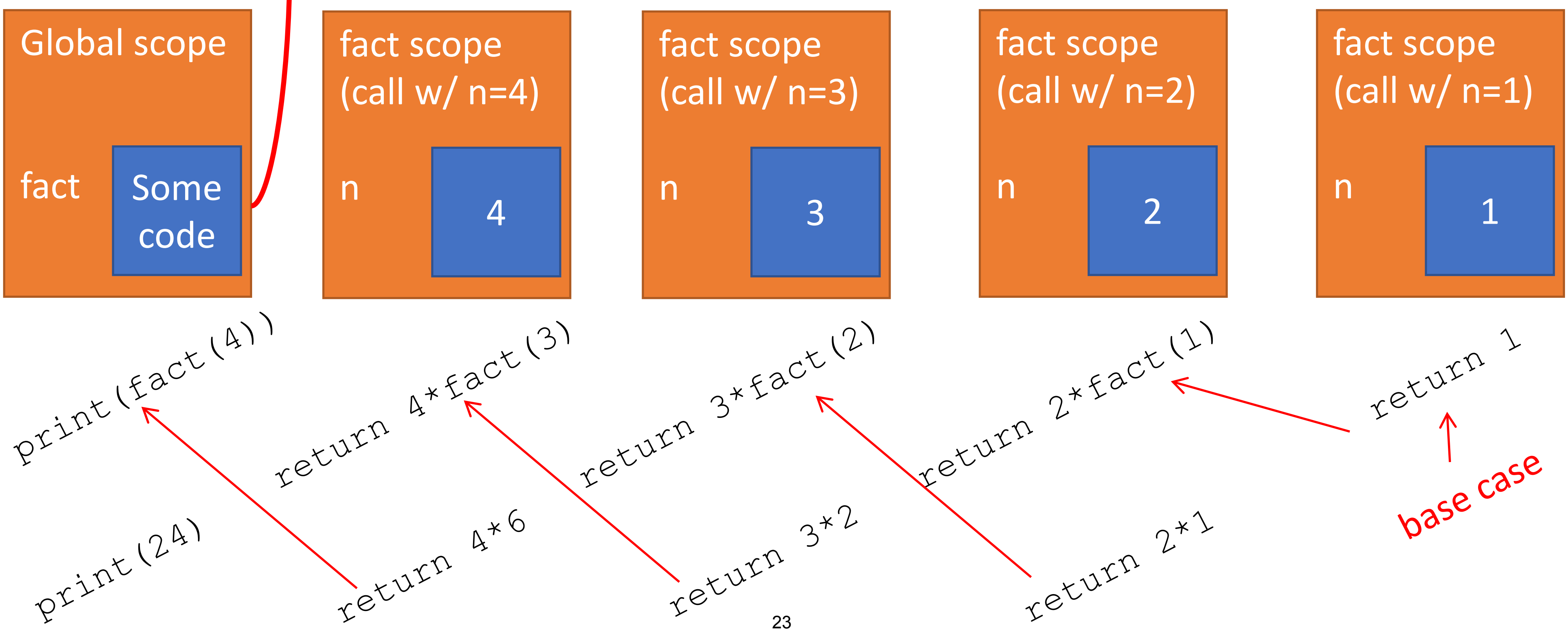
  n = 1 →
  ```
  if n == 1:
      return 1
  ```
  base case

- How to reduce problem? Rewrite in terms of something simpler to reach base case

  n*(n-1)! →
  ```
  else:
      return n*fact(n-1)
  ```
  recursive step

RECURSIVE FUNCTION SCOPE EXAMPLE

```
def fact(n):
    if n == 1:
        return 1
    else:
        return n*fact(n-1)

print(fact(4))
```



BIG IDEA

In recursion, each function call is completely separate.

Separate scope/environments.

Separate variable names.

Fully I-N-D-E-P-E-N-D-E-N-T

SOME OBSERVATIONS
Python Tutor LINK for factorial

- Each recursive call to a function creates its **own scope/environment**
- **Bindings of variables** in a scope are not changed by recursive call to same function
- Values of variable binding **shadow bindings** in other frames
- Flow of control passes back to **previous scope** once function call returns value

Using the same variable names but they are different objects in separate scopes

ITERATION vs. RECURSION

This version is much more Pythonic!

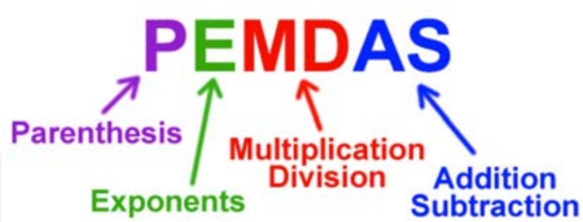
```
def factorial_iter(n):
    prod = 1
    for i in range(1,n+1):
        prod *= i
    return prod
```

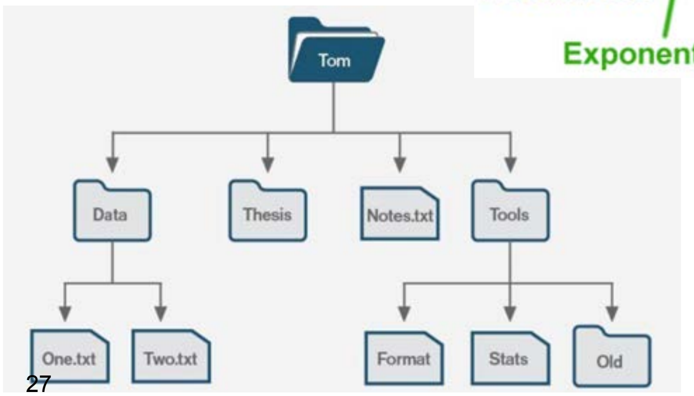
```
def fact_recur(n):
    if n == 1:
        return 1
    else:
        return n*fact_recur(n-1)
```

- Recursion may be efficient from programmer POV
- Recursion may not be efficient from computer POV

WHEN to USE RECURSION? SO FAR WE SAW VERY SIMPLE CODE

- **Multiplication of two numbers** did not need a recursive function, did not even need an iterative function!
- Factorial was a little more intuitive to implement with recursion
  - We translated a mathematical equation that told us the structure
- **MOST problems do not need recursion** to solve them
  - If iteration is more intuitive for you then solve them using loops!
- **SOME problems yield far simpler code** using recursion
  - Searching a file system for a specific file
  - Evaluating mathematical expressions that use parens for order of ops





SUMMARY

- Recursion is a
  - Programming method
  - Way to divide and conquer
- A **function calls itself**
- A problem is broken down into a **base case** and a **recursive step**
- A base case
  - **Something you know**
  - You'll **eventually reach** this case (if not, you have infinite recursion)
- A recursive step
  - The **same problem**
  - Just **slightly different** in a way that will eventually reach the base case

PYTHON CLASSES

(download slides and .py files to follow along)

6.100L Lecture 17

Ana Bell

OBJECTS

- Python supports many different kinds of data

```
1234      3.14159      "Hello"      [1, 5, 7, 11, 13]
{"CA": "California", "MA": "Massachusetts"}
```

- Each is an **object**, and every object has:
  - An internal **data representation** (primitive or composite)
  - A set of procedures for **interaction** with the object
- An object is an **instance** of a **type**
  - `1234` is an instance of an `int`
  - `"hello"` is an instance of a `str`

OBJECT ORIENTED PROGRAMMING (OOP)

- **EVERYTHING IN PYTHON IS AN OBJECT** (and has a type)
- Can **create new objects** of some type
- Can **manipulate objects**
- Can **destroy objects**
  - Explicitly using `del` or just "forget" about them
  - Python system will reclaim destroyed or inaccessible objects – called "garbage collection"

WHAT ARE OBJECTS?

- Objects are **a data abstraction** that captures…

(1) An **internal representation**

- Through data attributes

(2) An **interface** for interacting with object

- Through methods (aka procedures/functions)
- Defines behaviors but hides implementation

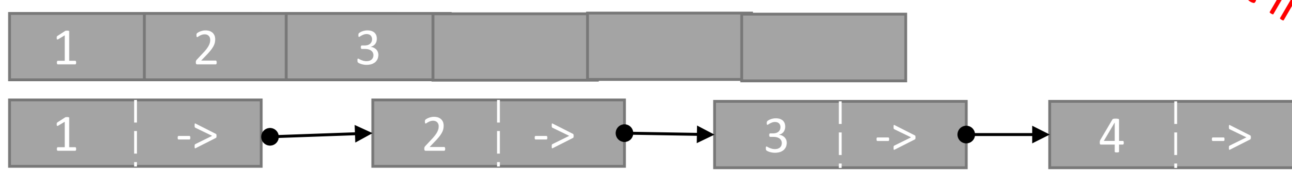
EXAMPLE:
[1,2,3,4] has type list

follow pointer to the next index

- (1) How are lists **represented internally**?
  Does not matter for so much for us as users (private representation)

L =

or **L =**



- (2) How to **interface with, and manipulate,** lists?
  - `L[i], L[i:j], +`
  - `len(), min(), max(), del(L[i])`
  - `L.append(),L.extend(),L.count(),L.index(), L.insert(),L.pop(),L.remove(),L.reverse(), L.sort()`
- Internal representation should be private
- Correct behavior may be compromised if you manipulate internal representation directly

REAL-LIFE EXAMPLES

- **Elevator**: a box that can change floors
  - Represent using length, width, height, max_capacity, current_floor
  - Move its location to a different floor, add people, remove people
- **Employee**: a person who works for a company
  - Represent using name, birth_date, salary
  - Can change name or salary
- **Queue at a store**: first customer to arrive is the first one helped
  - Represent customers as a list of str names
  - Append names to the end and remove names from the beginning
- **Stack of pancakes**: first pancake made is the last one eaten
  - Represent stack as a list of str
  - Append pancake to the end and remove from the end

ADVANTAGES OF OOP

- **Bundle data into packages** together with procedures that work on them through well-defined interfaces
- **Divide-and-conquer** development
  - Implement and test behavior of each class separately
  - Increased modularity reduces complexity
- **Classes** make it easy to **reuse** code
  - Many Python modules define new classes
  - Each class has a separate environment (no collision on function names)
  - Inheritance allows subclasses to redefine or extend a selected subset of a superclass' behavior

BIG IDEA

You write the class so you make the design decisions.

You decide what data represents the class.

You decide what operations a user can do with the class.

CREATING AND USING YOUR OWN TYPES WITH CLASSES

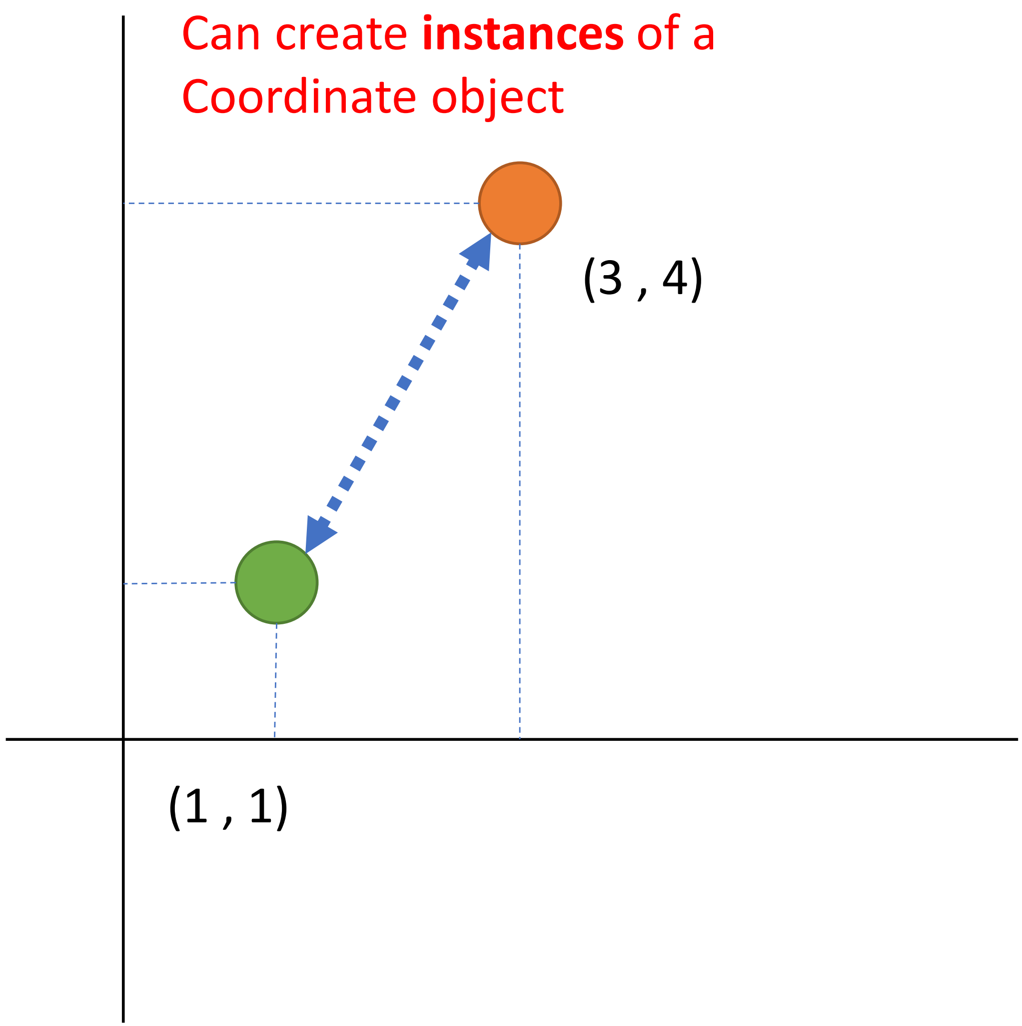
- Make a distinction between **creating a class** and **using an instance** of the class
- **Creating** the class involves
  - Defining the class name
  - Defining class attributes
  - *for example, someone wrote code to implement a list class*
- **Using** the class involves
  - Creating new **instances** of the class
  - Doing operations on the instances
  - *for example,* `L=[1,2]` *and* `len(L)`

A PARALLEL with FUNCTIONS

- **Defining a class** is like defining a function
  - With functions, we tell Python this procedure exists
  - With classes, we tell Python about a **blueprint for this new data type**
    - Its data attributes
    - Its procedural attributes

- **Creating instances of objects** is like calling the function
  - With functions we make calls with different actual parameters
  - With classes, we **create new object tinstances in memory of this type**
    - L1 = [1,2,3]
      L2 = [5,6,7]

COORDINATE TYPE DESIGN DECISIONS



- Decide what **data** elements constitute an object
  - In a 2D plane
  - A coordinate is defined by an **x and y value**

- Decide **what to do** with coordinates
  - Tell us how far away the coordinate is on the x or y axes
  - Measure the **distance** between two coordinates, Pythagoras

DEFINE YOUR OWN TYPES

- Use the `class` keyword to define a new type

class definition name/type class parent

```
class Coordinate(object):
    #define attributes here
```

- Similar to `def`, indent code to indicate which statements are part of the **class definition**
- The word `object` means that `Coordinate` is a Python object and **inherits** all its attributes (will see in future lects)

WHAT ARE ATTRIBUTES?

- Data and procedures that "**belong**" to the class
- **Data attributes**
  - Think of data as other objects/variables that make up the class
  - *for example, a coordinate is made up of two numbers*
- **Methods** (procedural attributes)
  - Think of methods as functions that only work with this class
  - How to interact with the object
  - *for example you can define a distance between two coordinate objects but there is no meaning to a distance between two list objects*

DEFINING HOW TO CREATE AN INSTANCE OF A CLASS

- First have to define **how to create an instance** of class
- Use a **special method called `__init__`** to initialize some data attributes or perform initialization operations

```
class Coordinate(object):
    def __init__(self, xval, yval):
        self.x = xval
        self.y = yval
```

what data initializes a Coordinate object

special method to create an instance is double __ underscore

parameter to refer to an instance of the class without having created one yet

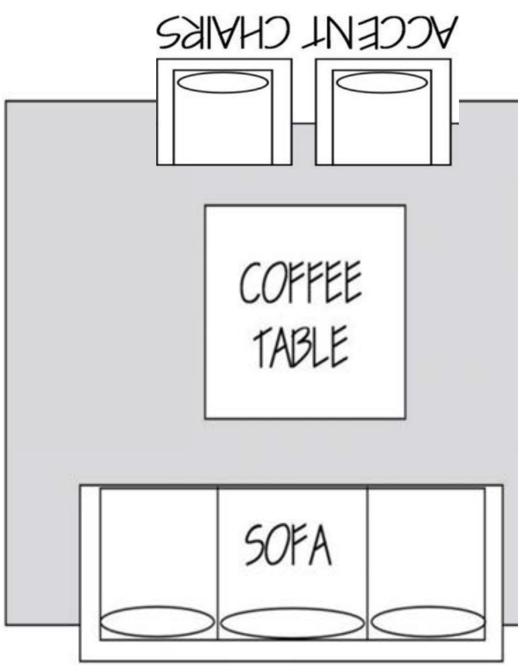
two data attributes make up your type

- `self` allows you to create **variables that belong to this object**
- Without `self`, you are just creating regular variables!

WHAT is self? ROOM EXAMPLE

self is the blueprint's way of accessing attributes (data and methods)

- Think of the class definition as a **blueprint** with placeholders for actual items
  - self has a chair
  - self has a coffee table
  - self has a sofa



- Now when you create **ONE** instance (name it living_room), self becomes this actual object
  - living_room has a blue chair
  - living_room has a black table
  - living_room has a white sofa
- Can make **many instances** using the same blueprint

BIG IDEA

When defining a class, we don’t have an actual tangible object here.

It’s only a definition.

Recall the __init__ method in the class def:

```
def __init__(self, xval, yval):
        self.x = xval
        self.y = yval
```

ACTUALLY CREATING AN INSTANCE OF A CLASS

- Don't provide argument for `self`, Python does this automatically

```
c = Coordinate(3,4)
origin = Coordinate(0,0)
```

create a new object of type `Coordinate` and pass in 3 and 4 to the `__init__`

- Data attributes of an instance are called **instance variables**
  - Data attributes were defined with `self.XXX` and they are accessible with dot notation for the lifetime of the object
  - All instances have these data attributes, but with different values!

```
print(c.x)
print(origin.x)
```

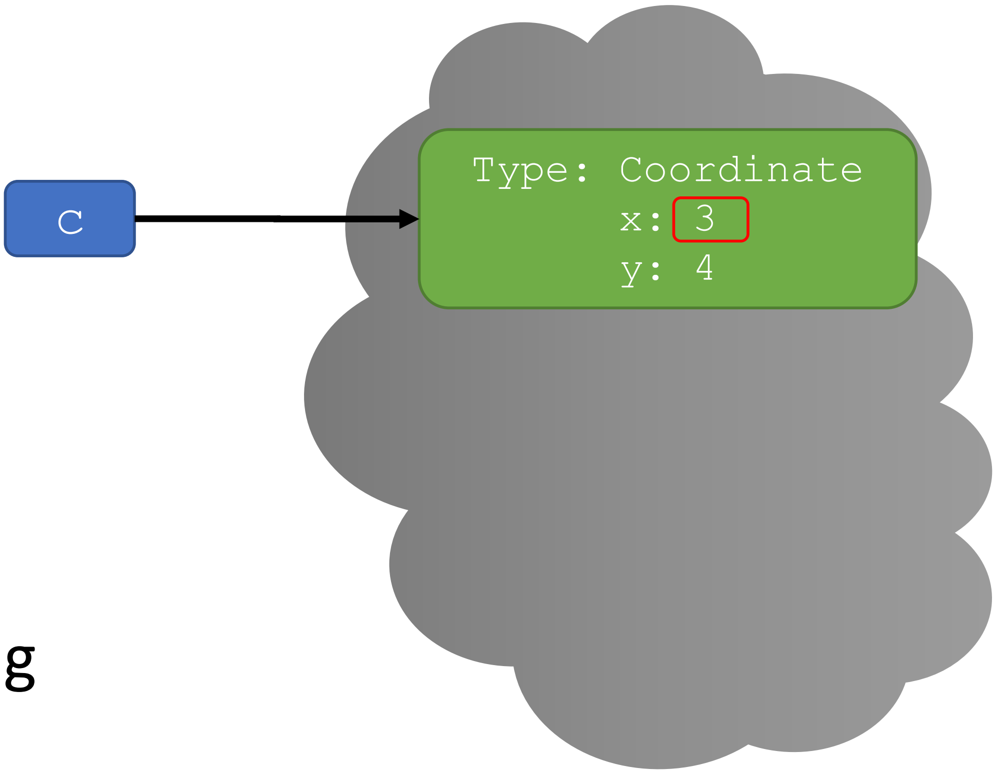
use the dot notation to access an attribute of instance `c`

VISUALIZING INSTANCES

- Suppose we create an instance of a coordinate

```
c = Coordinate(3,4)
```

- Think of this as creating a structure in memory
- Then evaluating
  `c.x`
  looks up the structure to which `c` points, then finds the binding for `x` in that structure

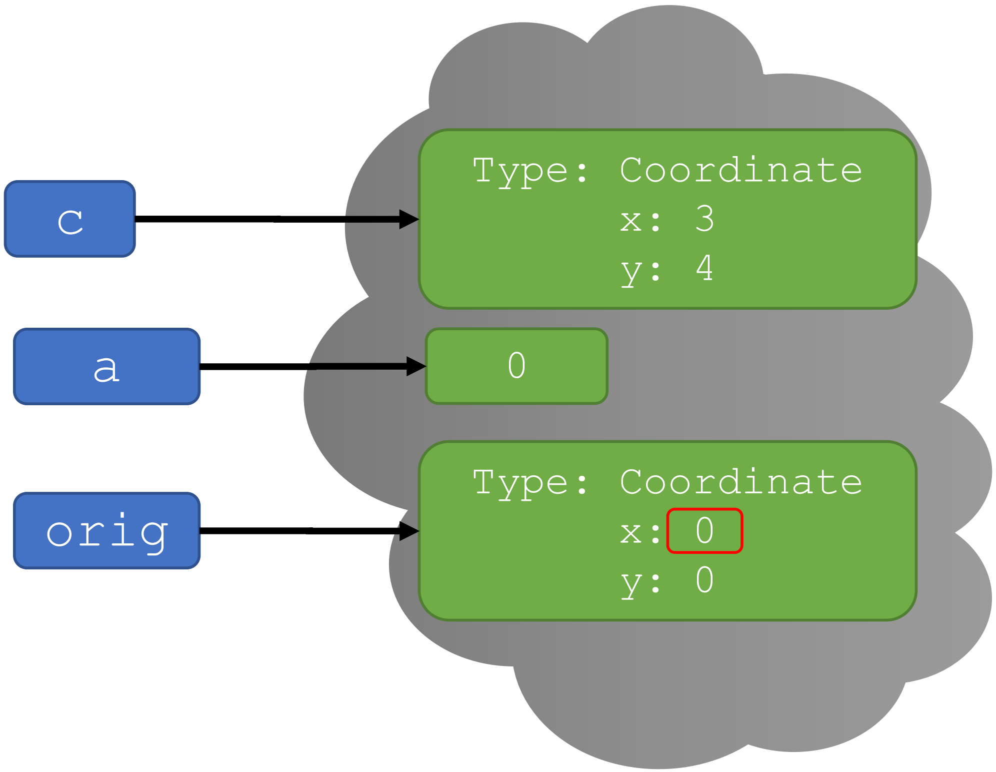


VISUALIZING INSTANCES: in memory

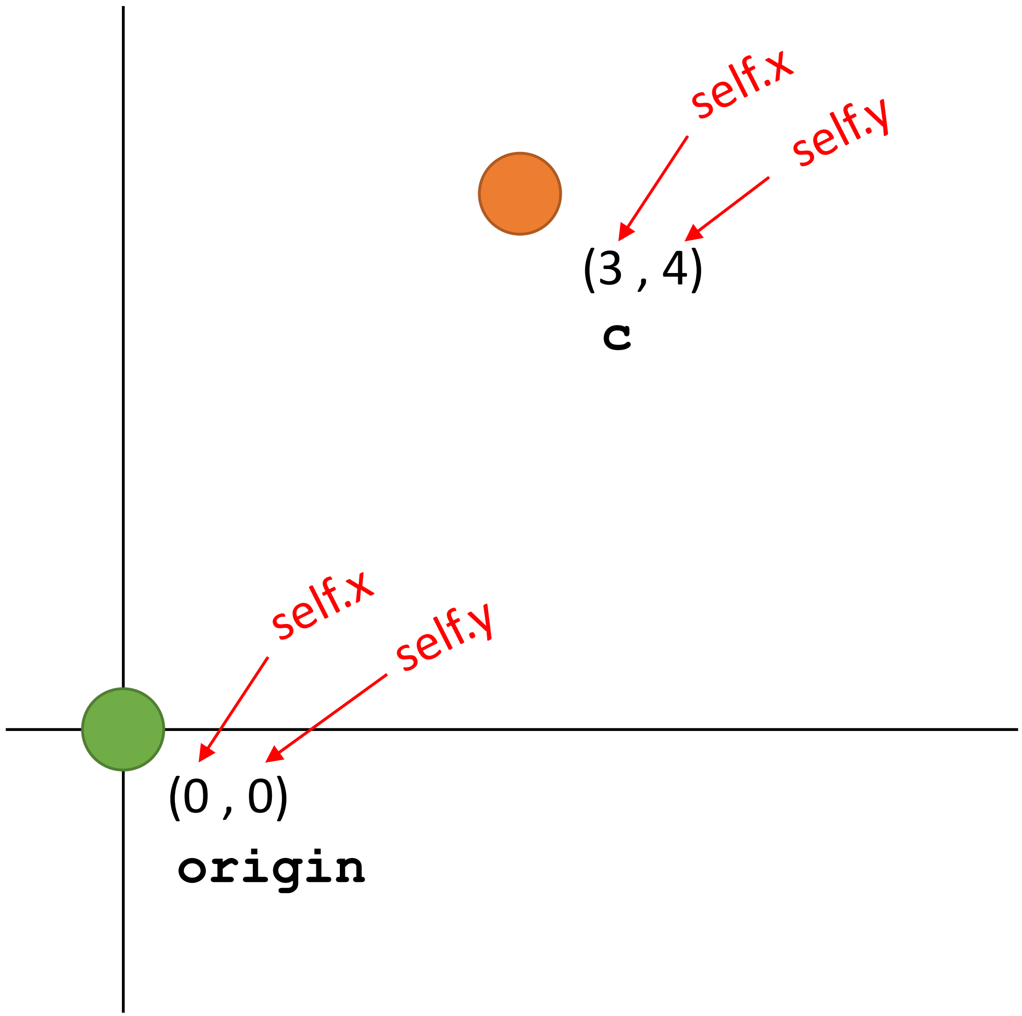
- Make another instance using a variable

```
a = 0
orig = Coordinate(a,a)
orig.x
```

- All these are just objects in memory!
- We just access attributes of these objects



VISUALIZING INSTANCES: draw it



The template for a Coordinate type

```
class Coordinate(object):
    def __init__(self, xval, yval):
        self.x = xval
        self.y = yval
```

```
c = Coordinate(3,4)
origin = Coordinate(0,0)
print(c.x)
print(origin.x)
```

Code to make actual tangible Coordinate objects (aka instances)

WHAT IS A METHOD?

- Procedural attribute
  - Think of it like a **function that works only with this class**
- Python always passes the object as the first argument
  - Convention is to use **`self`** as the name of the first argument of all methods

DEFINE A METHOD FOR THE `Coordinate` CLASS

```
class Coordinate(object):
    def __init__(self, xval, yval):
        self.x = xval
        self.y = yval
    def distance(self, other):
        x_diff_sq = (self.x-other.x)**2
        y_diff_sq = (self.y-other.y)**2
        return (x_diff_sq + y_diff_sq)**0.5
```

use it to refer to the obj I call this method on

another parameter to method

dot notation to access x of self

dot notation to access x of other

- Other than `self` and dot notation, methods behave just like functions (take params, do operations, return)

HOW TO CALL A METHOD?

- The "." **operator** is used to access any attribute
  - A data attribute of an object (we saw `c.x`)
  - A method of an object
- Dot notation

```
<object_variable>.<method>(<parameters>)
```

Object to call method on, becomes self in the class def

Name of method

Not including self. `self` is the obj before the dot!

- Familiar?

```
my_list.append(4)
my_list.sort()
```

HOW TO USE A METHOD

Recall the definition of distance method:

```
def distance(self, other):
        x_diff_sq = (self.x-other.x)**2
        y_diff_sq = (self.y-other.y)**2
        return (x_diff_sq + y_diff_sq)**0.5
```

Using the class:

```
c = Coordinate(3,4)
orig = Coordinate(0,0)
print(c.distance(orig))
```

object to call method on

name of method

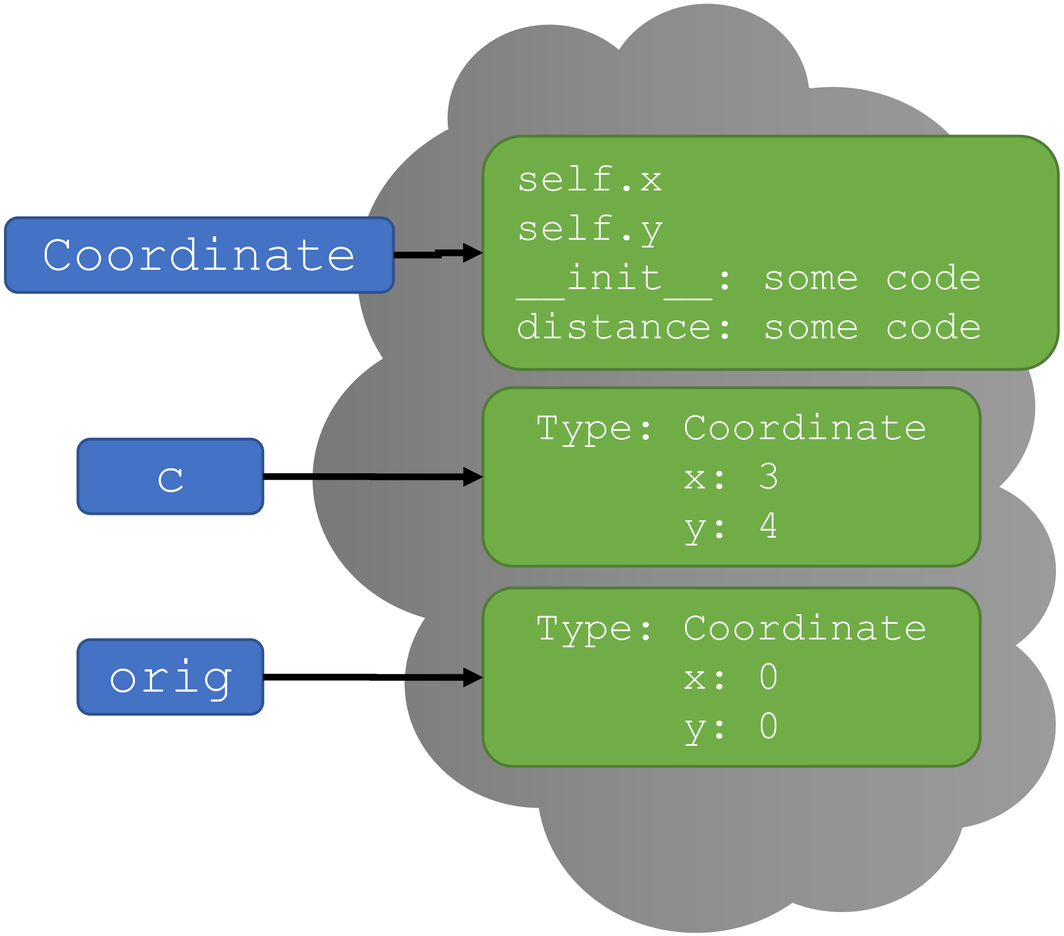
parameters not including `self` (`self` is implied to be `c`)

- Notice that `self` becomes the object you call the method on (the thing before the dot!)

VISUALIZING INVOCATION

- Coordinate class is an object in memory
  - From the class definition
- Create two Coordinate objects

```
c = Coordinate(3,4)
orig = Coordinate(0,0)
```

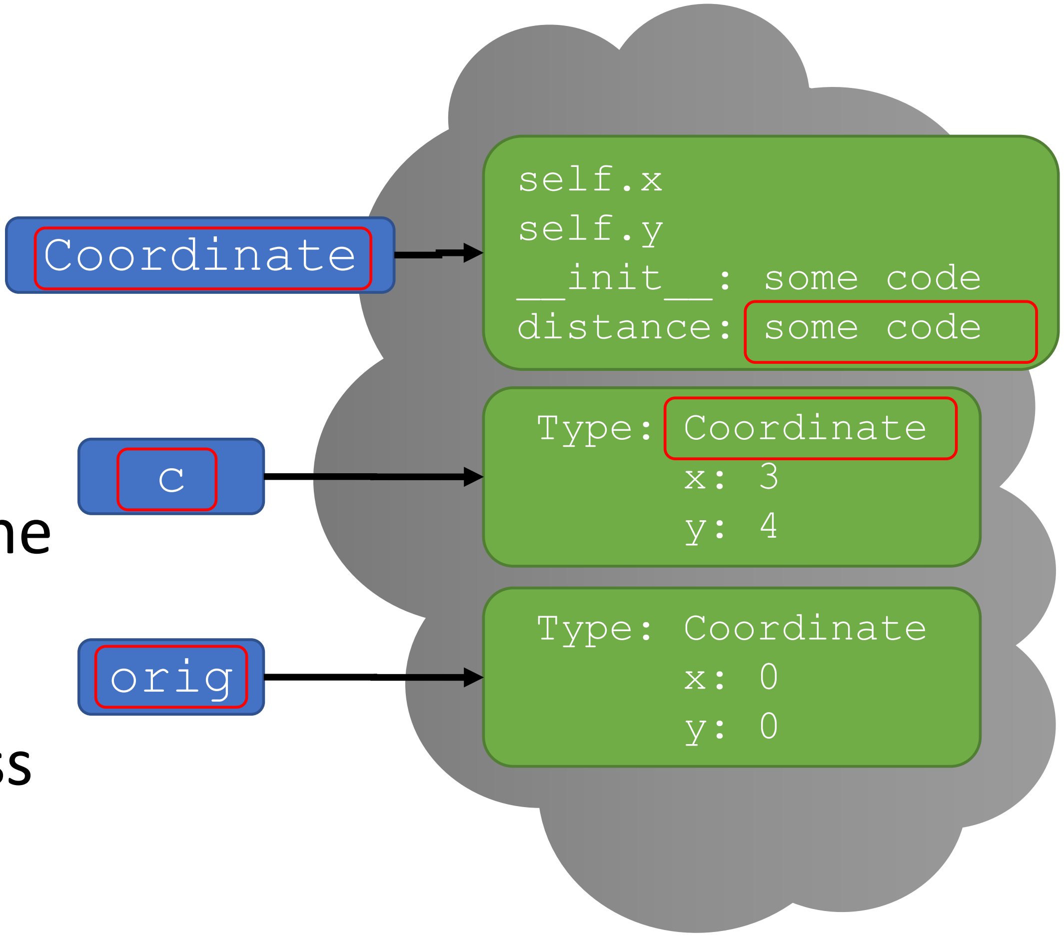


VISUALIZING INVOCATION

- Evaluate the method call

`c.distance(orig)`

- 1) The object is before the dot
- 2) Looks up the type of `c`
- 3) The method to call is after the dot.
- 4) Finds the binding for `distance` in that object class
- 5) Invokes that method with
  `c` as `self` and
  `orig` as `other`



HOW TO USE A METHOD

- Conventional way

```
c = Coordinate(3,4)
zero = Coordinate(0,0)
c.distance(zero)
```

object to call method on, this is self in the class def

name of method

parameters not including `self` (`self` is implied to be `c`)

- Equivalent to

```
c = Coordinate(3,4)
zero = Coordinate(0,0)
Coordinate.distance(c, zero)
```

name of class (NOT an object of type Coordinate)

name of method

parameters, including an object to call the method on, representing `self`

BIG IDEA

The . operator accesses either data attributes or methods.

Data attributes are defined with `self`.something

Methods are functions defined inside the class with `self` as the first parameter.

THE POWER OF OOP

- **Bundle together objects** that share
  - Common attributes and
  - Procedures that operate on those attributes
- Use **abstraction** to make a distinction between how to implement an object vs how to use the object
- Build **layers** of object abstractions that inherit behaviors from other classes of objects
- Create our **own classes of objects** on top of Python's basic classes

MORE PYTHON CLASS METHODS

(download slides and .py files to follow along)

6.100L Lecture 18

Ana Bell

IMPLEMENTING THE CLASS vs USING THE CLASS

- Write code from two different perspectives

Implementing a new object type with a class

- **Define** the class
- Define **data attributes** (WHAT IS the object)
- Define **methods** (HOW TO use the object)

Class abstractly captures **common** properties and behaviors

Using the new object type in code

- Create **instances** of the object type
- Do **operations** with them

Instances have **specific values** for attributes

RECALL THE COORDINATE CLASS

- Class **definition** tells Python the **blueprint** for a type Coordinate

```
class Coordinate(object):
    """ A coordinate made up of an x and y value """
    def __init__(self, x, y):
        """ Sets the x and y values """
        self.x = x
        self.y = y
    def distance(self, other):
        """ Returns euclidean dist between two Coord obj """
        x_diff_sq = (self.x-other.x)**2
        y_diff_sq = (self.y-other.y)**2
        return (x_diff_sq + y_diff_sq)**0.5
```

ADDING METHODS TO THE COORDINATE CLASS

- Methods are functions that **only work with objects of this type**

```
class Coordinate(object):
    """ A coordinate made up of an x and y value """
    def __init__(self, x, y):
        """ Sets the x and y values """
        self.x = x
        self.y = y
    def distance(self, other):
        """ Returns euclidean dist between two Coord obj """
        x_diff_sq = (self.x-other.x)**2
        y_diff_sq = (self.y-other.y)**2
        return (x_diff_sq + y_diff_sq)**0.5
    def to_origin(self):
        """ always sets self.x and self.y to 0,0 """
        self.x = 0
        self.y = 0
```

MAKING COORDINATE INSTANCES

- Creating **instances** makes actual Coordinate **objects in memory**
- The objects can be **manipulated**
  - Use **dot notation** to call methods and access data attributes

x data attr has a value of 3
y data attr has a value of 4

```
c = Coordinate(3,4)
origin = Coordinate(0,0)

print(f"c's x is {c.x} and origin's x is {origin.x}")
print(c.distance(origin))

c.to_origin()
print(c.x, c.y)
```

Method didn't return anything,
just set c's x and y to 0.

CLASS DEFINITION OF AN OBJECT TYPE vs INSTANCE OF A CLASS

- Class name is the **type**

  `class Coordinate(object)`

- Class is defined generically
  - Use `self` to refer to some instance while defining the class

  `(self.x – self.y)**2`

  - `self` is a parameter to methods in class definition

- Class defines data and methods **common across all instances**

- Instance is **one specific object**

  `coord = Coordinate(1,2)`

- Data attribute values vary between instances

  `c1 = Coordinate(1,2)`

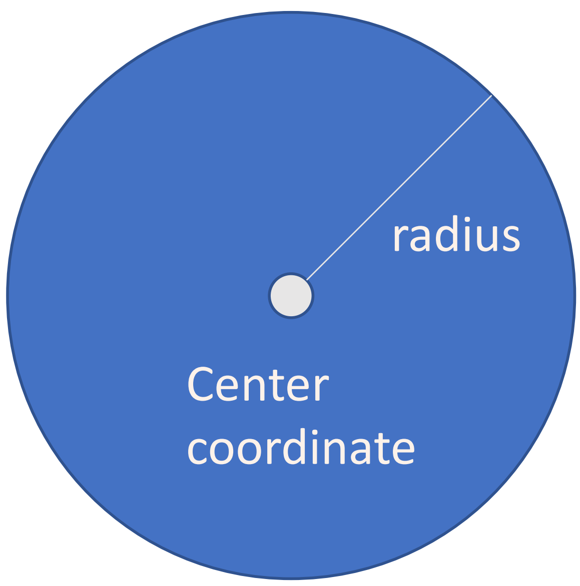
  `c2 = Coordinate(3,4)`

  - `c1` and `c2` have different data attribute values `c1.x` and `c2.x` because they are different objects

- Instance has the **structure of the class**

USING CLASSES TO BUILD OTHER CLASSES

- Example: use Coordinates to build Circles
- Our implementation will use **2 data attributes**
  - Coordinate object representing the center
  - int object representing the radius



CIRCLE CLASS:
DEFINITION and INSTANCES

Will be a Coordinate object

Will be an int

```
class Circle(object):
    def __init__(self, center, radius):
        self.center = center
        self.radius = radius
```

Data attributes, do not need to have the same names as params

```
center = Coordinate(2, 2)
my_circle = Circle(center, 2)
```

YOU TRY IT!

- Add code to the init method to check that the type of center is a Coordinate obj and the type of radius is an int. If either are not these types, raise a ValueError.

```
def __init__(self, center, radius):
    self.center = center
    self.radius = radius
```

CIRCLE CLASS: DEFINITION and INSTANCES

```
class Circle(object):
    def __init__(self, center, radius):
        self.center = center
        self.radius = radius
    def is_inside(self, point):
        """ Returns True if point is in self, False otherwise """
        return point.distance(self.center) < self.radius
```

self is a Circle object

point is a Coordinate object

Coordinate object

Method called on a Coordinate obj

Coordinate object

```
center = Coordinate(2, 2)
my_circle = Circle(center, 2)
p = Coordinate(1,1)
print(my_circle.is_inside(p))
```

Method that only works with obj of type Circle

Coordinate object

Circle object

YOU TRY IT!

- Are these two methods in the Circle class functionally equivalent?

```
class Circle(object):
    def __init__(self, center, radius):
        self.center = center
        self.radius = radius

    def is_inside1(self, point):
        return point.distance(self.center) < self.radius

    def is_inside2(self, point):
        return self.center.distance(point) < self.radius
```

EXAMPLE: FRACTIONS

- Create a **new type** to represent a number as a fraction
- **Internal representation** is two integers
  - Numerator
  - Denominator
- **Interface** a.k.a. **methods** a.k.a **how to interact** with `Fraction` objects
  - Add, subtract
  - Invert the fraction
- Let’s write it together!

NEED TO CREATE INSTANCES

```
class SimpleFraction(object):
    def __init__(self, n, d):
        self.num = n
        self.denom = d
```

MULTIPLY FRACTIONS

```
class SimpleFraction(object):
    def __init__(self, n, d):
        self.num = n
        self.denom = d
    def times(self, oth):
        top = self.num*oth.num
        bottom = self.denom*oth.denom
        return top/bottom
```

SimpleFraction objects so they each have
* num
* denom

Access num or denom to do the math

ADD FRACTIONS

```
class SimpleFraction(object):
    def __init__(self, n, d):
        self.num = n
        self.denom = d
    .........
    def plus(self, oth):
        top = self.num*oth.denom + self.denom*oth.num
        bottom = self.denom*oth.denom
        return top/bottom
```

LET'S TRY IT OUT

```
f1 = SimpleFraction(3, 4)
f2 = SimpleFraction(1, 4)
print(f1.num)
print(f1.denom)
print(f1.plus(f2))
print(f1.times(f2))
```

3

4

1.0

0.1875

YOU TRY IT!

- Add two methods to invert fraction object according to the specs below:

```
class SimpleFraction(object):
    """ A number represented as a fraction """
    def __init__(self, num, denom):
        self.num = num
        self.denom = denom
    def get_inverse(self):
        """ Returns a float representing 1/self """
        pass
    def invert(self):
        """ Sets self's num to denom and vice versa.
            Returns None. """
        pass

# Example:
f1 = SimpleFraction(3,4)
print(f1.get_inverse())   # prints 1.33333333 (note this one returns value)
f1.invert()               # acts on data attributes internally, no return
print(f1.num, f1.denom)   # prints 4 3
```

LET'S TRY IT OUT WITH MORE THINGS

```
f1 = SimpleFraction(3, 4)
f2 = SimpleFraction(1, 4)
print(f1.num)              → 3
print(f1.denom)            → 4
print(f1.plus(f2))         → 1.0
print(f1.times(f2))        → 0.1875
```

What if we want to keep as a fraction?

And what if we want to have print and * work as we would expect?

```
print(f1)
print(f1 * f2)
```

<__main__.SimpleFraction object at 0x00000234A8C41DF0>

Error!

SPECIAL OPERATORS IMPLEMENTED WITH DUNDER METHODS

- +, -, ==, <, >, len(), print, and many others are shorthand notations
- Behind the scenes, these **get replaced by a method**!

https://docs.python.org/3/reference/datamodel.html#basic-customization

- Can **override** these to work with your class

SPECIAL OPERATORS IMPLEMENTED WITH DUNDER METHODS

- Define them with **double underscores** before/after

`__add__(self, other)`	→	`self + other`
`__sub__(self, other)`	→	`self - other`
`__mul__(self, other)`	→	**`self * other`**
`__truediv__(self, other)`	→	`self / other`
`__eq__(self, other)`	→	`self == other`
`__lt__(self, other)`	→	`self < other`
`__len__(self)`	→	`len(self)`
`__str__(self)`	→	**`print(self)`**
`__float__(self)`	→	**`float(self) i.e cast`**
`__pow__`	→	`self**other`

... and others

PRINTING OUR OWN DATA TYPES

PRINT REPRESENTATION OF AN OBJECT

```
>>> c = Coordinate(3,4)
>>> print(c)
<__main__.Coordinate object at 0x7fa918510488>
```

- **Uninformative** print representation by default
- Define a **`__str__` method** for a class
- Python calls the `__str__` method when used with `print` on your class object
- You choose what it does! Say that when we print a `Coordinate` object, want to show

```
>>> print(c)
<3,4>
```

DEFINING YOUR OWN PRINT METHOD

```
class Coordinate(object):
    def __init__(self, xval, yval):
        self.x = xval
        self.y = yval
    def distance(self, other):
        x_diff_sq = (self.x-other.x)**2
        y_diff_sq = (self.y-other.y)**2
        return (x_diff_sq + y_diff_sq)**0.5
    def __str__(self):
        return "<"+str(self.x)+","+str(self.y)+">"
```

name of special method

must **return a string**

WRAPPING YOUR HEAD AROUND TYPES AND CLASSES

- Can ask for the type of an object instance

```
>>> c = Coordinate(3,4)
>>> print(c)
<3,4>
>>> print(type(c))
<class __main__.Coordinate>
```

Return of the __str__ method

The type of object c is a class Coordinate

- This makes sense since

```
>>> print(Coordinate)
<class __main__.Coordinate>
>>> print(type(Coordinate))
<type 'type'>
```

A Coordinate is a class

A Coordinate class is a type of object

- Use `isinstance()` to check if an object is a `Coordinate`

```
>>> print(isinstance(c, Coordinate))
True
```

EXAMPLE: FRACTIONS WITH DUNDER METHODS

- Create a **new type** to represent a number as a fraction
- **Internal representation** is two integers
    - Numerator
    - Denominator
- **Interface** a.k.a. **methods** a.k.a **how to interact** with `Fraction` objects
    - Add, sub, mult, div to work with +, -, *, /
    - Print representation, convert to a float
    - Invert the fraction
- Let’s write it together!

CREATE & PRINT INSTANCES

```
class Fraction(object):
    def __init__(self, n, d):
        self.num = n
        self.denom = d
    def __str__(self):
        return str(self.num) + "/" + str(self.denom)
```

Concatenation means that every piece has to be a str

LET'S TRY IT OUT

```
f1 = Fraction(3, 4)
f2 = Fraction(1, 4)
f3 = Fraction(5, 1)
print(f1)
print(f2)
print(f3)
```

→ 3/4

→ 1/4

→ 5/1

Ok, but looks weird!

YOU TRY IT!

- Modify the str method to represent the Fraction as just the numerator, when the denominator is 1. Otherwise its representation is the numerator then a / then the denominator.

```
class Fraction(object):
    def __init__(self, num, denom):
        self.num = num
        self.denom = denom
    def __str__(self):
        return str(self.num) + "/" + str(self.denom)

# Example:
a = Fraction(1,4)
b = Fraction(3,1)
print(a)      # prints 1/4
print(b)      # prints 3
```

IMPLEMENTING
+ - * /
float()

COMPARING METHOD vs. DUNDER METHOD

```
class SimpleFraction(object):
  def __init__(self, n, d):
    self.num = n
    self.denom = d
    .........
  def times(self, oth):
    top = self.num*oth.num
    bottom = self.denom*oth.denom
    return top/bottom
```

When we use this method, Python evaluates and returns this expression, which creates a float

```
class Fraction(object):
  def __init__(self, n, d):
    self.num = n
    self.denom = d
    .........
  def __mul__(self, other):
    top = self.num*other.num
    bottom = self.denom*other.denom
    return Fraction(top, bottom)
```

Note: we are creating and returning a new **instance of a Fraction**

LETS TRY IT OUT

```
a = Fraction(1,4)
b = Fraction(3,4)
print(a)                    1/4
c = a * b
print(c)                    3/16
```

Calls the __mul__method behind the scenes. This method returns Fraction(3,16)

Uses __str__ for a Fraction object

CLASSES CAN HIDE DETAILS

- These are all equivalent

```
print(a * b)
```

```
print(a.__mul__(b))
```

```
print(Fraction.__mul__(a, b))
```

Shorthand (nice and clear!)

Call to dunder method, bad style with dunder methods!

Explicit class call, passing in val for self, bad style in general!

- Every operation in Python comes back to a method call
- The first instance makes clear the operation, without worrying about the internal details! **Abstraction at work**

BIG IDEA

Special operations we’ve been using are just methods behind the scenes.

Things like:
print, len
+, *, -, /, <, >, <=, >=, ==, !=
[]
and many others!

CAN KEEP BOTH OPTIONS BY ADDING A METHOD TO CAST TO A `float`

```
class Fraction(object):
    def __init__(self, n, d):
        self.num = n
        self.denom = d
    .........
    def __float__(self):
        return self.num/self.denom
```

A float since it does the division directly

```
c = a * b
print(c)           → 3/16
print(float(c))    → 0.1875
```

Repr for Fraction(3,16)

LETS TRY IT OUT SOME MORE

```
a = Fraction(1,4)
b = Fraction(2,3)
c = a * b
print(c)
```

→ 2/12

- Not quite what we might expect? It's not reduced.
- Can we fix this?

ADD A METHOD

```
class Fraction(object):
   .........
   def reduce(self):
      def gcd(n, d):
         while d != 0:
            (d, n) = (n%d, d)
         return n
      if self.denom == 0:
         return None
      elif self.denom == 1:
         return self.num
      else:
         greatest_common_divisor = gcd(self.num, self.denom)
         top = int(self.num/greatest_common_divisor)
         bottom = int(self.denom/greatest_common_divisor)
         return Fraction(top, bottom)
```

Function to find the greatest common divisor

Call it inside the method

Still want a Fraction object back

```
c = a*b
print(c)
print(c.reduce())
```

2/12

1/6

WE HAVE SOME IMPROVEMENTS TO MAKE

```
class Fraction(object):
  ............
  def reduce(self):
    def gcd(n, d):
      while d != 0:
        (d, n) = (n%d, d)
      return n
    if self.denom == 0:
      return None
    elif self.denom == 1:
      return self.num
    else:
      greatest_common_divisor = gcd(self.num, self.denom)
      top = int(self.num/greatest_common_divisor)
      bottom = int(self.denom/greatest_common_divisor)
      return Fraction(top, bottom)
```

Is this a good idea?
It does not return a Fraction so
can no longer add or multiply
this by other Fractions

CHECK THE TYPES, THEY'RE DIFFERENT

```
a = Fraction(4,1)
b = Fraction(3,9)
ar = a.reduce()            → 4
br = b.reduce()            → 1/3
print(ar, type(ar))        → 4 <class 'int'>
print(br, type(br))        → 1/3 <class '__main__.Fraction'>
c = ar * br
```

Error! It's trying to multiply an **int** with a **Fraction**.
We never defined how to do this – only a Fraction with another Fraction

YOU TRY IT!

- Modify the code to return a Fraction object when denominator is 1

```
class Fraction(object):
  def reduce(self):
    def gcd(n, d):
      while d != 0:
        (d, n) = (n%d, d)
      return n
    if self.denom == 0:
      return None
    elif self.denom == 1:
      return self.num
    else:
      greatest_common_divisor = gcd(self.num, self.denom)
      top = int(self.num/greatest_common_divisor)
      bottom = int(self.denom/greatest_common_divisor)
      return Fraction(top, bottom)

# Example:
f1 = Fraction(5,1)
print(f1.reduce())    # prints 5/1 not 5
```

WHY OOP and BUNDLING THE DATA IN THIS WAY?

- Code is **organized** and **modular**
- Code is easy to **maintain**
- It's easy to **build upon** objects to make more complex objects

- **Decomposition and abstraction** at work with Python classes
  - Bundling data and behaviors means you can use objects consistently
  - Dunder methods are abstracted by common operations, but they're just methods behind the scenes!

INHERITANCE

(download slides and .py files to follow along)

6.100L Lecture 19

Ana Bell

WHY USE OOP AND CLASSES OF OBJECTS?

- Mimic real life
- Group different objects part of the same type

Instance:
Jelly
1 year old
brown

Instance:
Tiger
2 years old
brown

Instance:
Bean
0 years old
black

Instance:
5 years old
brown

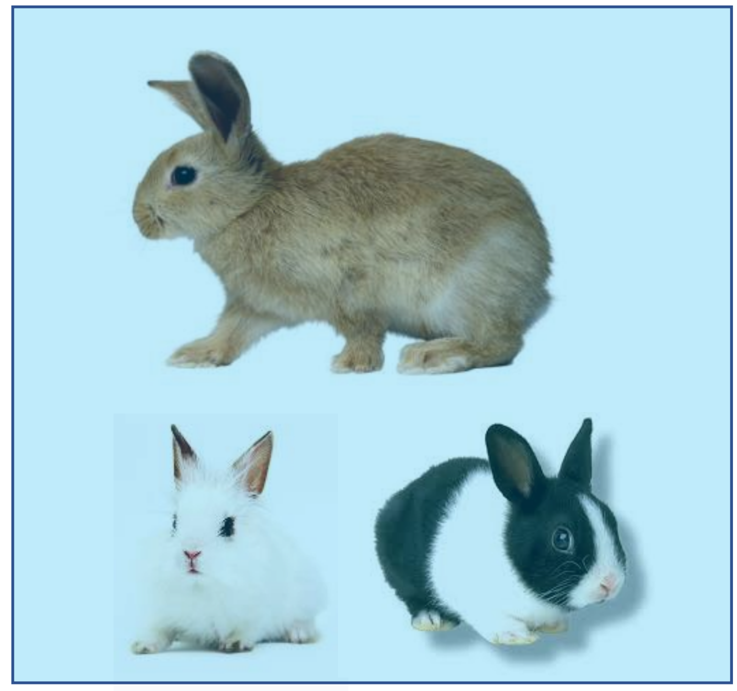
Instance:
2 years old
white

Instance:
1 year old
b/w

WHY USE OOP AND CLASSES OF OBJECTS?

- Mimic real life
- Group different objects part of the same type

All instances of a type have the same data abstraction and behaviors



GROUPS OF OBJECTS HAVE ATTRIBUTES (RECAP)

- **Data attributes**
  - How can you represent your object with data?
  - **What it is**

    for a coordinate: x and y values

    for an animal: age
- **Procedural attributes** (behavior/operations/**methods**)
  - How can someone interact with the object?
  - **What it does**

    for a coordinate: find distance between two

    for an animal: print how long it's been alive

HOW TO DEFINE A CLASS (RECAP)

class definition
name
class parent
Variable to refer to an instance of the class

```
class Animal(object):
    def __init__(self, age):
        self.age = age
        self.name = None
```

What data initializes an Animal type

Special method to create an instance

name is a data attribute even though an instance is not initialized with it as a param

```
myanimal = Animal(3)
```

One instance

Mapped to self.age in class def

GETTER AND SETTER METHODS

```
class Animal(object):
    def __init__(self, age):
        self.age = age
        self.name = None
    def __str__(self):
        return "animal:"+str(self.name)+":"+str(self.age)
```

- **Getters and setters** should be used outside of class to access data attributes

GETTER AND SETTER METHODS

```
class Animal(object):
    def __init__(self, age):
        self.age = age
        self.name = None
    def __str__(self):
        return "animal:"+str(self.name)+":"+str(self.age)
```

getter

```
    def get_age(self):
        return self.age
    def get_name(self):
        return self.name
```

setter

```
    def set_age(self, newage):
        self.age = newage
    def set_name(self, newname=""):
        self.name = newname
```

- **Getters and setters** should be used outside of class to access data attributes

AN INSTANCE and DOT NOTATION (RECAP)

- Instantiation creates an **instance of an object**

```
a = Animal(3)
```

- **Dot notation** used to access attributes (data and methods) though it is better to use getters and setters to access data attributes

```
a.age
```

```
a.get_age()
```

- access data attribute
- allowed, but not recommended

- access method
- best to use getters and setters

INFORMATION HIDING

- Author of class definition may **change data attribute** variable names

Replaced age data attribute by years

```
class Animal(object):
    def __init__(self, age):
        self.years = age
    def get_age(self):
        return self.years
```

- If you are **accessing data attributes** outside the class and class **definition changes**, may get errors
- Outside of class, use getters and setters instead
- Use `a.get_age()` NOT `a.age`
    - good style
    - easy to maintain code
    - prevents bugs

CHANGING INTERNAL REPRESENTATION

```
class Animal(object):
    def __init__(self, age):
        self.years = age
        self.name = None
    def __str__(self):
        return "animal:"+str(self.name)+":"+str(self.age)
    def get_age(self):
        return self.years
    def set_age(self, newage):
        self.years = newage
```

Change internal rep from self.age = age

```
a.get_age()    # works
a.age          # error
```

Accessing methods works correctly, but accessing data attributes no longer works.

- **Getters and setters** should be used outside of class to access data attributes

PYTHON NOT GREAT AT INFORMATION HIDING

- Allows you to **access data** from outside class definition

```
print(a.age)
```

- Allows you to **write to data** from outside class definition

```
a.age = 'infinite'
```

- Allows you to **create data attributes** for an instance from outside class definition

```
a.size = "tiny"
```

- It's **not good style** to do any of these!

USE OUR NEW CLASS

```
def animal_dict(L):
    """ L is a list
    Returns a dict, d, mappping an int to an Animal object.
    A key in d is all non-negative ints, n, in L. A value
    corresponding to a key is an Animal object with n as its age. """
    d = {}
    for n in L:
        if type(n) == int and n >= 0:
            d[n] = Animal(n)
    return d

L = [2,5,'a',-5,0]
```

Invoke the name of the class with parameter n (i.e. the age of that animal)

USE OUR NEW CLASS

- Python doesn't know how to call print recursively

```python
def animal_dict(L):
    """ L is a list
    Returns a dict, d, mappping an int to an Animal object.
    A key in d is all non-negative ints n L. A value corresponding
    to a key is an Animal object with n as its age. """
    d = {}
    for n in L:
        if type(n) == int and n >= 0:
            d[n] = Animal(n)
    return d

L = [2,5,'a',-5,0]

animals = animal_dict(L)
print(animals)
```

Return is a dict mapping int:Animal

```
{2: <__main__.Animal object at 0x00000199AFF350A0>,
 5: <__main__.Animal object at 0x00000199AFF35A30>,
 0: <__main__.Animal object at 0x00000199AFF35D00>}
```

USE OUR NEW CLASS

```
def animal_dict(L):
    """ L is a list
    Returns a dict, d, mappping an int to an Animal object.
    A key in d is all non-negative ints n L. A value corresponding
    to a key is an Animal object with n as its age. """
    d = {}
    for n in L:
        if type(n) == int and n >= 0:
            d[n] = Animal(n)
    return d

L = [2,5,'a',-5,0]

animals = animal_dict(L)
for n,a in animals.items():
    print(f'key {n} with val {a}')
```

Manually loop over animal objects and access their data attr through getter methods

```
key 2 with val animal:None:2
key 5 with val animal:None:5
key 0 with val animal:None:0
```

YOU TRY IT!

- Write a function that meets this spec.

```
def make_animals(L1, L2):
    """ L1 is a list of ints and L2 is a list of str
        L1 and L2 have the same length
    Creates a list of Animals the same length as L1 and L2.
    An animal object at index i has the age and name
    corresponding to the same index in L1 and L2, respectively. """

#For example:
L1 = [2,5,1]
L2 = ["blobfish", "crazyant", "parafox"]
animals = make_animals(L1, L2)
print(animals)      # note this prints a list of animal objects
for i in animals:   # this loop prints the individual animals
    print(i)
```

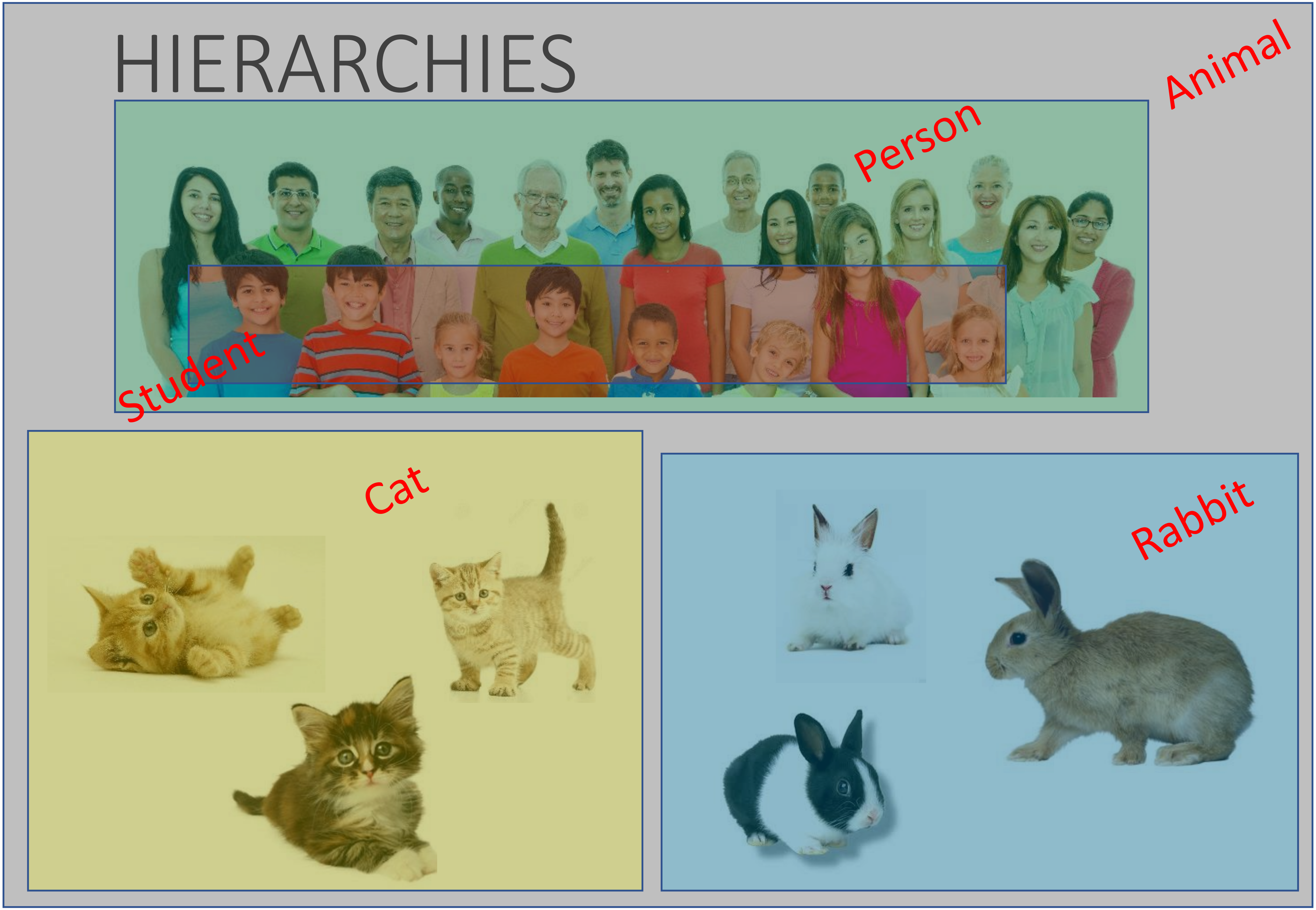
BIG IDEA

Access data attributes

(stuff defined by self.xxx)

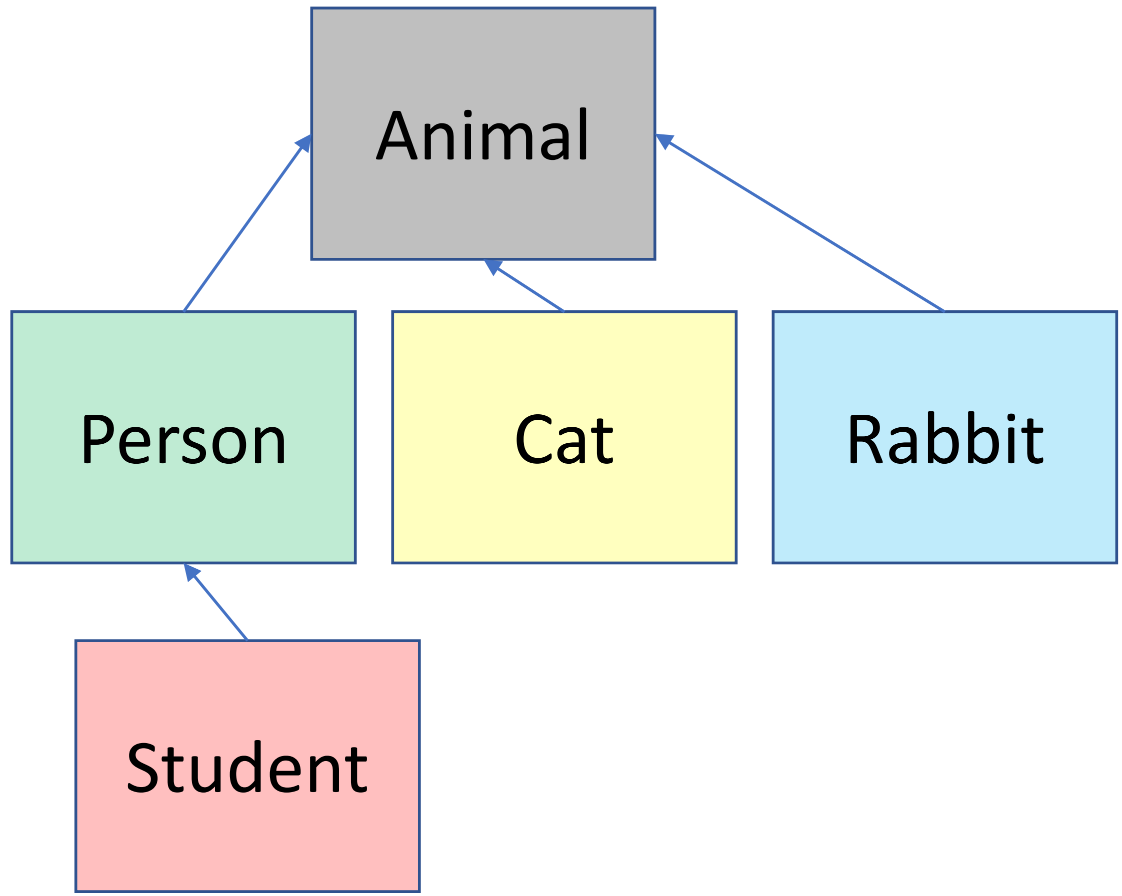
through methods – it's better style.

HIERARCHIES



HIERARCHIES

- **Parent class** (superclass)
- **Child class** (subclass)
  - **Inherits** all data and behaviors of parent class
  - **Add** more **info**
  - **Add** more **behavior**
  - **Override** behavior



INHERITANCE: PARENT CLASS

```
class Animal(object):
    def __init__(self, age):
        self.age = age
        self.name = None
    def get_age(self):
        return self.age
    def get_name(self):
        return self.name
    def set_age(self, newage):
        self.age = newage
    def set_name(self, newname=""):
        self.name = newname
    def __str__(self):
        return "animal:"+str(self.name)+":"+str(self.age)
```

- everything is an object
- class `object` implements basic operations in Python, like binding variables, etc

SUBCLASS CAT

INHERITANCE: SUBCLASS

Inherits all attributes of `Animal`:
__init__()
age, name
get_age(), get_name()
set_age(), set_name()
__str__()

```
class Cat(Animal):
    def speak(self):
        print("meow")
    def __str__(self):
        return "cat:"+str(self.name)+":"+str(self.age)
```

Add new functionality via speak method

Overrides __str__

- Add new functionality with `speak()`
  - Instance of type `Cat` can be called with new methods
  - Instance of type `Animal` throws error if called with `Cat`'s new method
- `__init__` is not missing, uses the `Animal` version

WHICH METHOD TO USE?

- Subclass can have **methods with same name** as superclass
- For an instance of a class, look for a method name in **current class definition**
- If not found, look for method name **up the hierarchy** (in parent, then grandparent, and so on)
- Use first method up the hierarchy that you found with that method name

SUBCLASS PERSON

```python
class Person(Animal):
    def __init__(self, name, age):
        Animal.__init__(self, age)
        self.set_name(name)
        self.friends = []
    def get_friends(self):
        return self.friends.copy()
    def add_friend(self, fname):
        if fname not in self.friends:
            self.friends.append(fname)
    def speak(self):
        print("hello")
    def age_diff(self, other):
        diff = self.age - other.age
        print(abs(diff), "year difference")
    def __str__(self):
        return "person:"+str(self.name)+":"+str(self.age)
```

Parent class is `Animal`

Call `Animal` constructor to run lines of code in Animal's init

Call `Animal`'s method

Add a new data attribute

New methods

Override `Animal`'s __str__ method

YOU TRY IT!

- Write a function according to this spec.

```
def make_pets(d):
    """ d is a dict mapping a Person obj to a Cat obj
    Prints, on each line, the name of a person, a colon, and the
    name of that person's cat """
    pass

p1 = Person("ana", 86)
p2 = Person("james", 7)
c1 = Cat(1)
c1.set_name("furball")
c2 = Cat(1)
c2.set_name("fluffsphere")

d = {p1:c1, p2:c2}
make_pets(d)   # prints ana:furball
               #        james:fluffsphere
```

BIG IDEA

A subclass can
use a parent's attributes,
override a parent's attributes, or
define new attributes.

Attributes are either data or methods.

SUBCLASS STUDENT

```python
import random

class Student(Person):
    def __init__(self, name, age, major=None):
        Person.__init__(self, name, age)
        self.major = major
    def change_major(self, major):
        self.major = major
    def speak(self):
        r = random.random()
        if r < 0.25:
            print("i have homework")
        elif 0.25 <= r < 0.5:
            print("i need sleep")
        elif 0.5 <= r < 0.75:
            print("i should eat")
        else:
            print("i'm still zooming")
    def __str__(self):
        return "student:"+str(self.name)+":"+str(self.age)+":"+str(self.major)
```

Bring in functions from `random` library

Inherits `Person` and `Animal` attributes

Person __init__ takes care of all initializations

Adds new data

- I looked up how to use the `random` library in the python docs
- `random()` method gives back float in [0, 1)

SUBCLASS RABBIT

CLASS VARIABLES AND THE `Rabbit` SUBCLASS

- **Class variables** and their values are shared between all instances of a class

```
class Rabbit(Animal):
    tag = 1
    def __init__(self, age, parent1=None,parent2=None):
        Animal.__init__(self, age)
        self.parent1 = parent1
        self.parent2 = parent2
        self.rid = Rabbit.tag
        Rabbit.tag += 1
```

parent class

Shared class variable

Access shared class variable

instance variable

Incrementing class variable changes it for all instances that may reference it

- `tag` used to give **unique id** to each new rabbit instance

RECALL THE `__init__` OF `Rabbit`

```
def __init__(self, age, parent1=None,parent2=None):
        Animal.__init__(self, age)
        self.parent1 = parent1
        self.parent2 = parent2
        self.rid = Rabbit.tag
        Rabbit.tag += 1
```

Shared across all instances

`Rabbit.tag` 2

`r1`

Age: 8
Parent1: None
Parent2: None
Rid: 1

`r1 = Rabbit(8)`

RECALL THE `__init__` OF `Rabbit`

```
def __init__(self, age, parent1=None,parent2=None):
        Animal.__init__(self, age)
        self.parent1 = parent1
        self.parent2 = parent2
        self.rid = Rabbit.tag
        Rabbit.tag += 1
```

Shared across all instances

`Rabbit.tag` 3

`r1`

Age: 8
Parent1: None
Parent2: None
Rid: 1

`r2`

Age: 6
Parent1: None
Parent2: None
Rid: 2

```
r1 = Rabbit(8)
r2 = Rabbit(6)
```

RECALL THE `__init__` OF `Rabbit`

```
def __init__(self, age, parent1=None,parent2=None):
        Animal.__init__(self, age)
        self.parent1 = parent1
        self.parent2 = parent2
        self.rid = Rabbit.tag
        Rabbit.tag += 1
```

Shared across all instances

```
r1 = Rabbit(8)
r2 = Rabbit(6)
r3 = Rabbit(10)
```

Rabbit.tag 4

r1
Age: 8
Parent1: None
Parent2: None
Rid: 1

r2
Age: 6
Parent1: None
Parent2: None
Rid: 2

r3
Age: 10
Parent1: None
Parent2: None
Rid: 3

Rabbit GETTER METHODS

```
class Rabbit(Animal):
    tag = 1
    def __init__(self, age, parent1=None, parent2=None):
        Animal.__init__(self, age)
        self.parent1 = parent1
        self.parent2 = parent2
        self.rid = Rabbit.tag
        Rabbit.tag += 1
    def get_rid(self):
        return str(self.rid).zfill(5)
    def get_parent1(self):
        return self.parent1
    def get_parent2(self):
        return self.parent2
```

Method on a string to pad the beginning with zeros for example, 00001 not 1

- getter methods specific for a `Rabbit` class
- there are also getters `get_name` and `get_age` inherited from `Animal`

WORKING WITH YOUR OWN TYPES

```
def __add__(self, other):
    # returning object of same type as this class
    return Rabbit(0, self, other)
```

recall Rabbit's `__init__(self, age, parent1=None, parent2=None)`

- Define **+ operator** between two `Rabbit` instances
  - Define what something like this does: `r4 = r1 + r2` where `r1` and `r2` are `Rabbit` instances
  - `r4` is a new `Rabbit` instance with age 0
  - `r4` has `self` as one parent and `other` as the other parent
  - In `__init__`, **`parent1` and `parent2` are of type `Rabbit`**

RECALL THE `__init__` OF `Rabbit`

```
def __init__(self, age, parent1=None,parent2=None):
        Animal.__init__(self, age)
        self.parent1 = parent1
        self.parent2 = parent2
        self.rid = Rabbit.tag
        Rabbit.tag += 1
```

Shared across all instances

`Rabbit.tag` 5

```
r1 = Rabbit(8)
r2 = Rabbit(6)
r3 = Rabbit(10)
r4 = r1 + r2
```

Rabbit(0,r1, r2)

`r1`
Age: 8
Parent1: None
Parent2: None
Rid: 1

`r2`
Age: 6
Parent1: None
Parent2: None
Rid: 2

`r3`
Age: 10
Parent1: None
Parent2: None
Rid: 3

`r4`
Age: 0
Parent1: obj bound to r1
Parent2: obj bound to r2
Rid: 4

SPECIAL METHOD TO COMPARE TWO `Rabbits`

- Decide that two rabbits are equal if they have the **same two parents**

```
def __eq__(self, other):
    parents_same = (self.p1.rid == oth.p1.rid and self.p2.rid == oth.p2.rid)
    parents_opp = (self.p2.rid == oth.p1.rid and self.p1.rid == oth.p2.rid)
    return parents_same or parents_opp
```

Booleans checking r1+r2 or r2+r1

- Compare ids of parents since **ids are unique** (due to class var)
- Note you can't compare objects directly
  - For ex. with `self.parent1 == other.parent1`
  - This calls the `__eq__` method over and over until call it on `None` and gives an `AttributeError` when it tries to do `None.parent1`

BIG IDEA

Class variables are shared between all instances.

If one instance changes it, it's changed for every instance.

OBJECT ORIENTED PROGRAMMING

- Create your own **collections of data**
- **Organize** information
- **Division** of work
- Access information in a **consistent** manner

- Add **layers** of complexity
  - Hierarchies
  - Child classes inherit data and methods from parent classes
- Like functions, classes are a mechanism for **decomposition** and **abstraction** in programming

6.100L Introduction to Computer Science and Programming Using Python
Fall 2022

FITNESS TRACKER OBJECT ORIENTED PROGRAMMING EXAMPLE

(download slides and .py files to follow along)

6.100L Lecture 20

Ana Bell

IMPLEMENTING THE CLASS vs USING THE CLASS

Implementing a new object type with a class

- **Define** the class
- Define **data attributes** (WHAT IS the object)
- Define **methods** (HOW TO use the object)

Class abstractly captures **common** properties and behaviors

Using the new object type in code

- Create **instances** of the object type
- Do **operations** with them

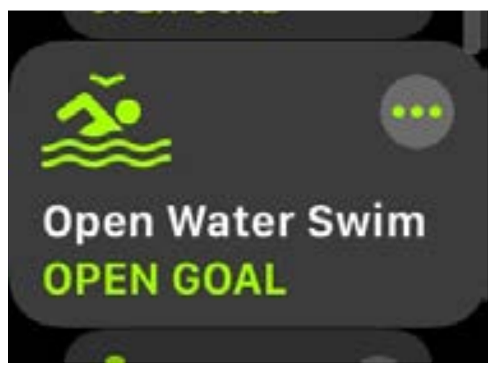
Instances have **specific values** for attributes

Two different coding perspectives

Workout Tracker Example

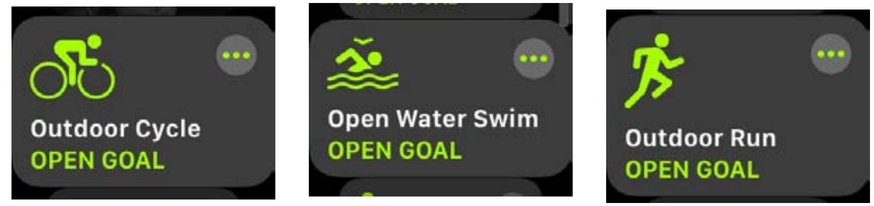
Thanks to Sam Madden for this OOP example (his slides have been modified)

- Suppose we are writing a program to track workouts, e.g., for a smart watch

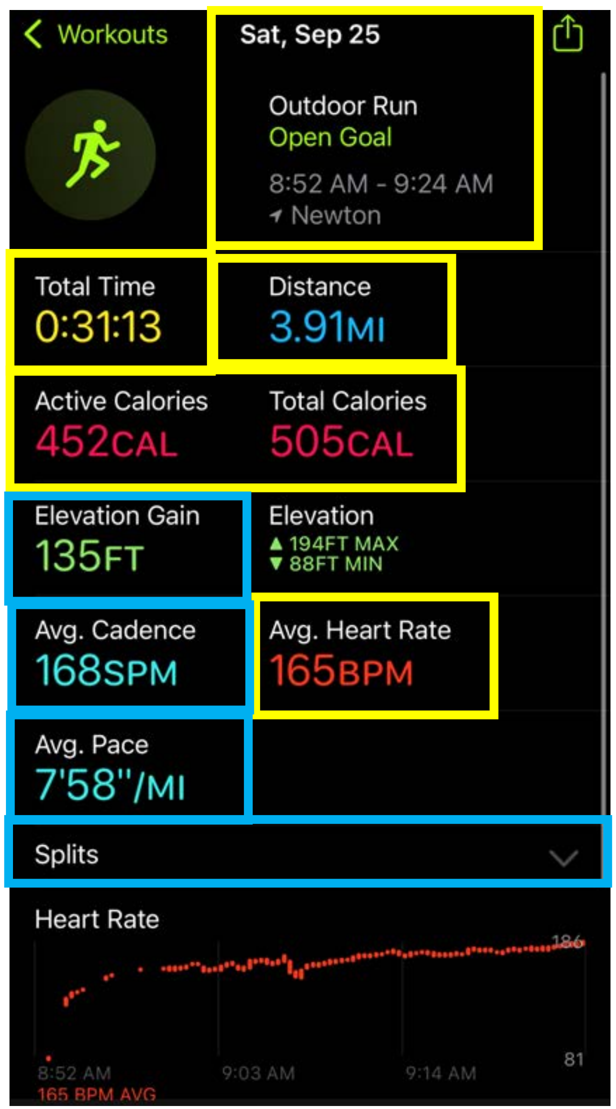


Different kinds of workouts

Fitness Tracker



Different types of workouts



Common properties:

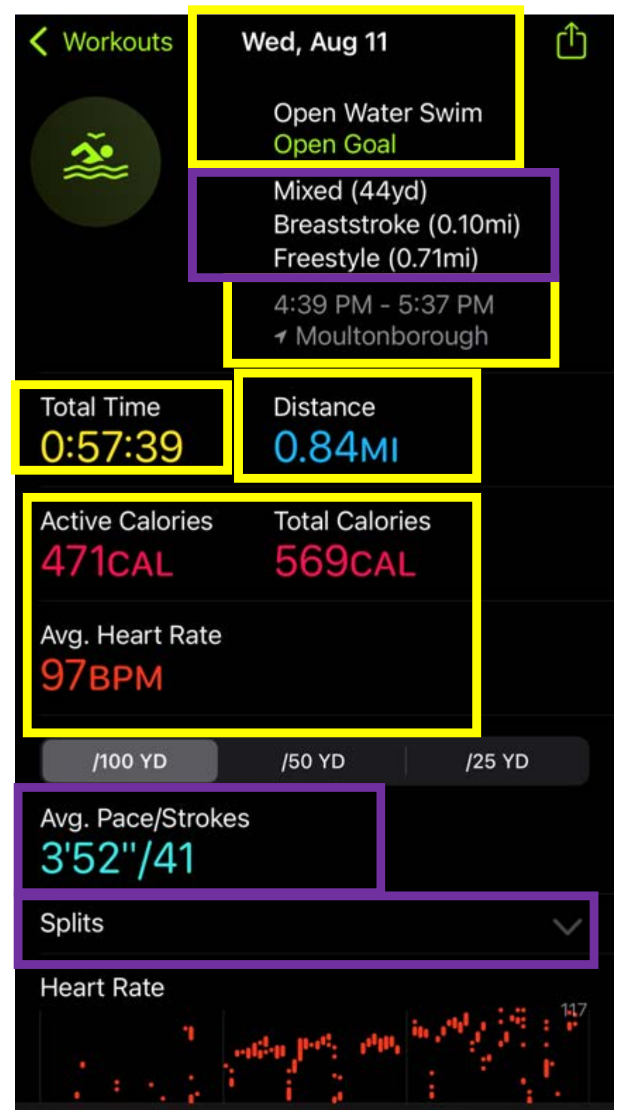
Icon	*Kind*
Date	*Start Time*
End Time	*Calories*
Heart Rate	Distance

Swimming Specific:

Swimming Pace
Stroke Type
100 yd Splits

Running Specific:

Cadence
Running Pace
Mile Splits
Elevation



GROUPS OF OBJECTS HAVE ATTRIBUTES (RECAP)

- **Data attributes**
  - How can you represent your object with data?
  - **What it is**
  - *for a coordinate: x and y values*
  - *for a workout: start time, end time, calories*
- **Functional attributes** (behavior/operations/**methods**)
  - How can someone interact with the object?
  - **What it does**
  - *for a coordinate: find distance between two*
  - *for a workout: display an information card*

DEFINE A SIMPLE CLASS (RECAP)

class definition

name

class parent

variable to refer to an instance of the class

Start and end time, and calories burned

```
class Workout(object):
    def __init__(self, start, end, calories):
        self.start = start
        self.end = end
        self.calories = calories
        self.icon = '😓'
        self.kind = 'Workout'
```

“constructor” - special method to create an instance

Icon and kind are attributes even though an instance is not initialized with them as a param (And python strings can contain emojis! 🤯)

```
my_workout = Workout('9/30/2021 1:35 PM', '9/30/2021 1:57 PM', 200)
```

one instance

Mapped to start, end, calories in constructor

GETTER AND SETTER METHODS (RECAP)

```
class Workout(object):
    def __init__(self, start, end, calories):
        self.start = start
        self.end = end
        self.calories = calories
        self.icon = '😓'
        self.kind = 'Workout'
    def get_calories(self):
        return self.calories
    def get_start(self):
        return self.start
    def get_end(self):
        return self.end
    def set_calories(self, calories):
        self.calories = calories
    def set_start(self, start):
        self.start = start
    def set_end(self, end):
        self.end = end
```

getter

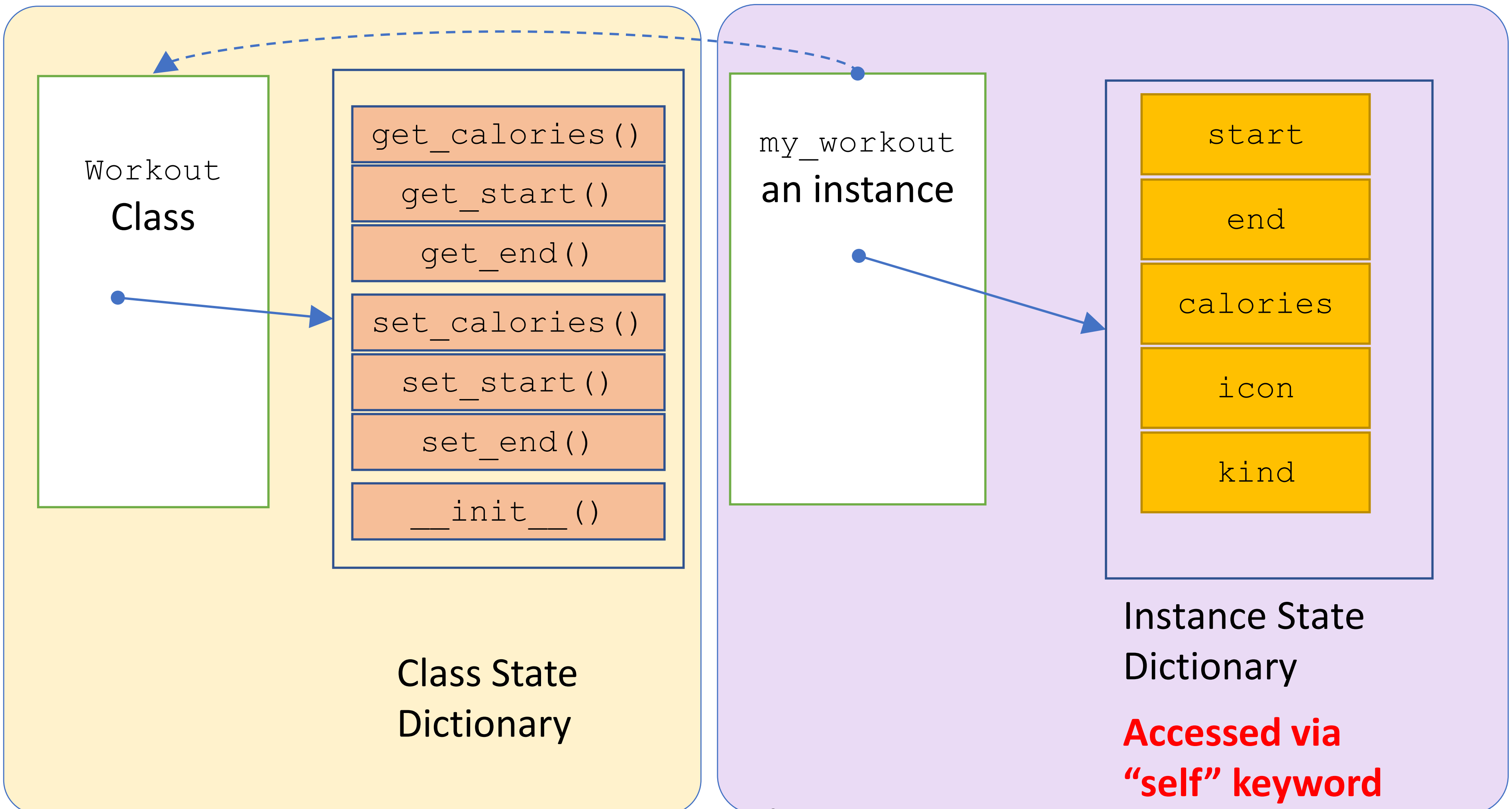
setter

Getters and setters used outside of class to access data attributes

Demo

SELF PROVIDES ACCESS TO CLASS STATE

```
my_workout = Workout('9/30/2021 1:35 PM', 9/30/2021 1:57 PM', 200)
```



AN INSTANCE and DOT NOTATION (RECAP)

- Instantiation creates an **instance of an object**

```
myWorkout = Workout('9/30/2021 1:35 PM', '9/30/2021 1:57 PM', 200)
```

- **Dot notation** used to access attributes (data and methods)
- It's better to use getters and setters to access data attributes

```
my_workout.calories
```

- access data attribute directly
- allowed, but not recommended

```
my_workout.get_calories()
```

- access attribute via method
- better, because it supports information hiding

WHY INFORMATION HIDING?

- Keep the **interface** of your class as **simple** as possible
- Use getters & setters, not attributes
  - i.e., `get_calories()` method NOT `calories` attribute
  - Prevents bugs due to changes in implementation
- May seem **inconsequential in small programs**, but for large programs complex interfaces increase the potential for bugs
- If you are writing a class for others to use, you are **committing to maintaining its interface**!

CHANGING THE CLASS IMPLEMENTATION

- Author of class definition may **change internal representation or implementation**
  - Use a class variable
  - Now `get_calories` estimates calories based of workout duration if calories are not passed in
- If **accessing data attributes** outside the class and class **implementation changes**, may get errors

CHANGING THE CLASS IMPLEMENTATION

Class variable – all instances of Workout can read this

Defaults to None if not passed in

self.start and self.end are objects of type datetime, not strings

```
class Workout:
    cal_per_hr = 200

    def __init__(self, start, end, calories=None):
        self.start = parser.parse(start)
        self.end = parser.parse(end)
        self.calories = calories # may be None
        self.icon = '😓'
        self.kind = 'Workout'

    def get_calories(self):
        if (calories == None):
            return Workout.cal_per_hr*(self.end-self.start).total_seconds()/3600
        else:
            return self.calories
```

If calories was not passed in, estimate based on elapsed time

Allowed on datetime objects

If calories was passed in, just use that value

ASIDE: `datetime` OBJECTS
OTHER PYTON LIBRARIES

- Takes the string representing the date and time and **converts it to a `datetime` object**

```
from dateutil import parser
start = '9/30/2021 1:35 PM'
end = '9/30/2021 1:45 PM'
start_date = parser.parse(start)
end_date = parser.parse(end)
type(start_date)
```

Brings in a bunch of functions and classes

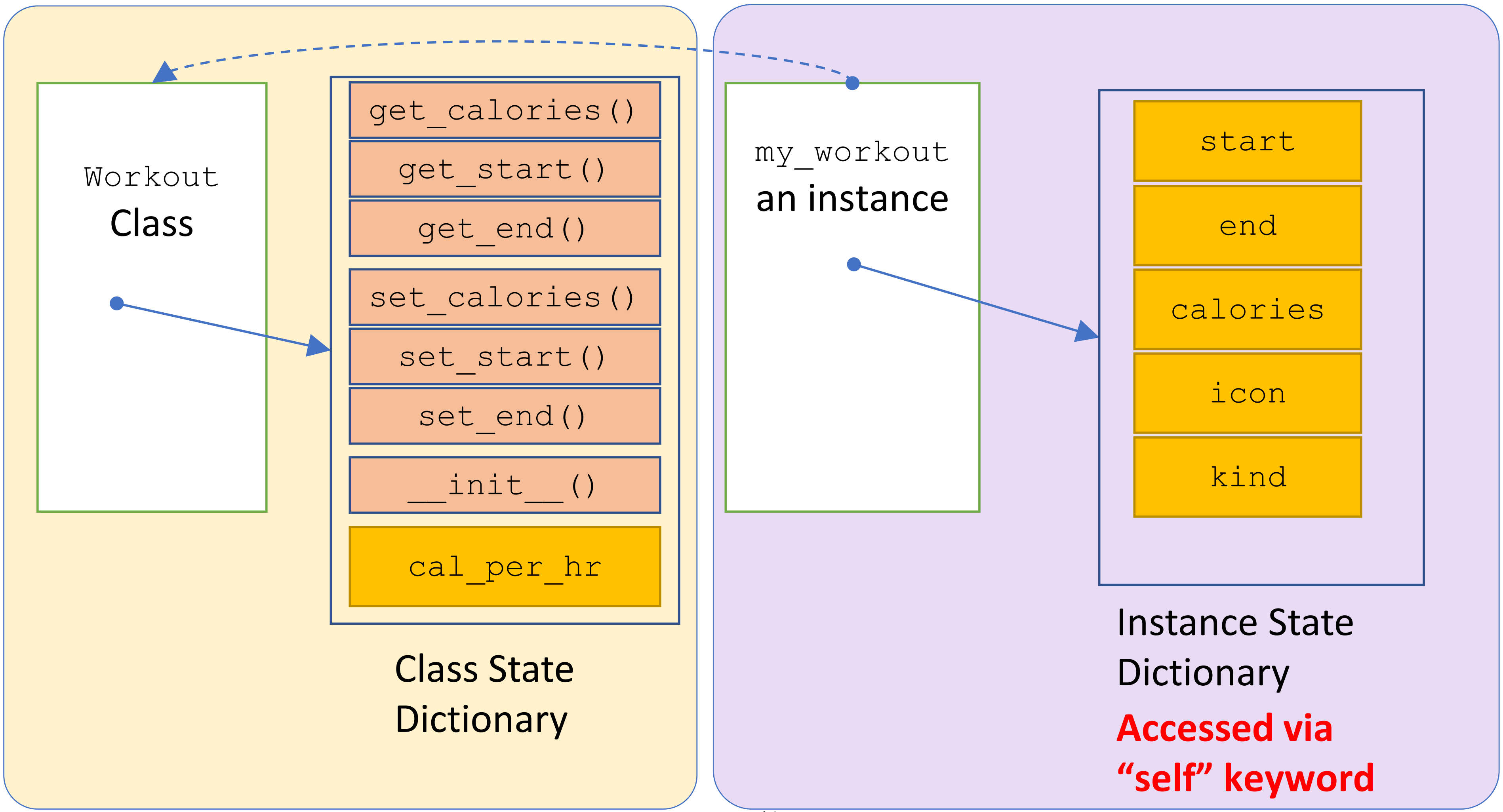
Type is `datetime.datetime`

- Why do this? Because it **makes operations with dates easy**! The datetime object takes care of everything

```
print((end_date-start_date).total_seconds())
```

Prints 600

CLASS VARIABLES LIVE IN CLASS STATE DICTIONARY



CLASS VARIABLES

- Associate a **class variable with all instances** of a class
- Warning: if an instance changes the class variable, it's changed for all instances

```
class Workout:
    cal_per_hr = 200
    def __init__(self, start, end, calories):
        …
```

cal_per_hr is set to 200 for all new instances of Workout

```
print(Workout.cal_per_hr)
```

No instance required, prints 200

```
w = Workout('1/1/2021 2:34', '1/1/2021 3:35', None)
print(w.cal_per_hr)
```

Prints 200

```
Workout.cal_per_hr = 250
print(w.cal_per_hr)
```

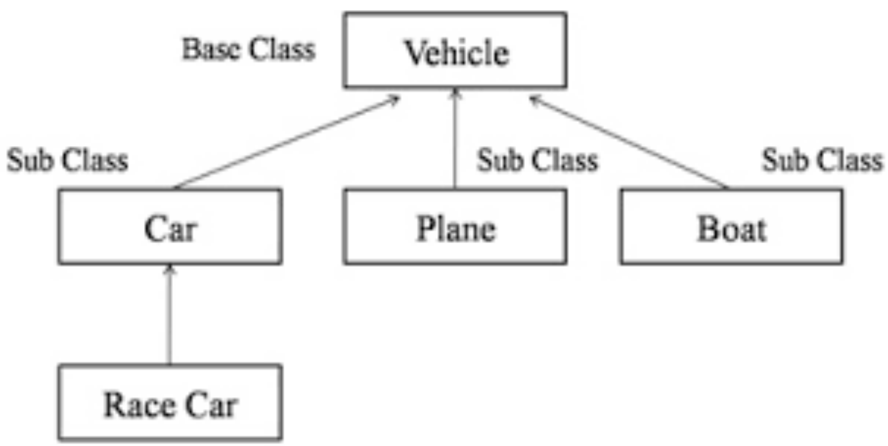
Prints 250

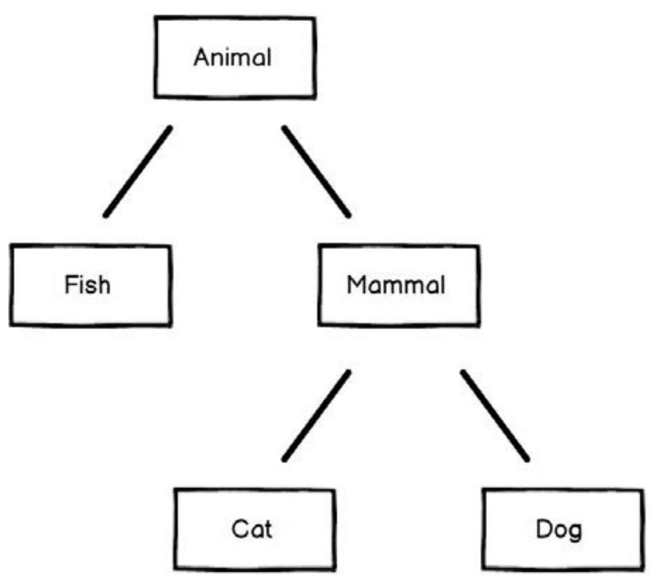
Bad style to change the class variable outside the class definition. Write a method to do it!

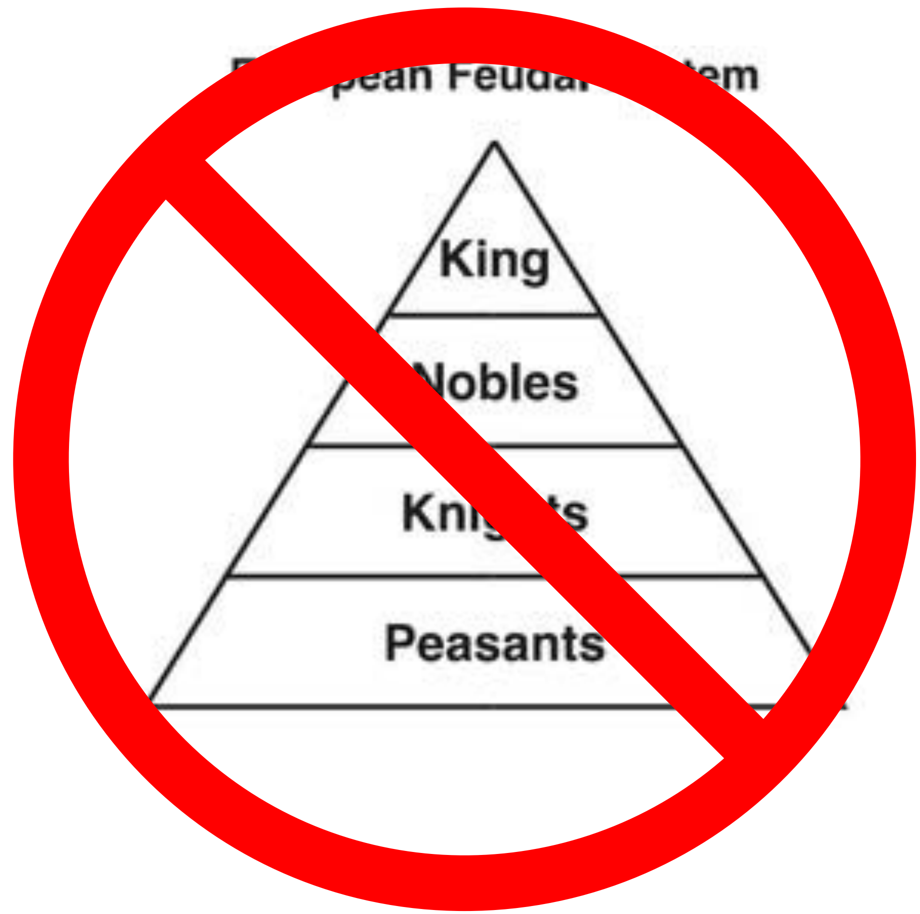
YOU TRY IT!

- Write lines of code to create two Workout objects.
  - One Workout object saved as variable `w_one`, from `Jan 1 2021 at 3:30 PM` until `4 PM`. You want to estimate the calories from this workout. Print the number of calories for `w_one`.
  - Another Workout object saved as `w_two`, from `Jan 1 2021 at 3:35 PM` until `4 PM`. You know you burned `300` calories for this workout. Print the number of calories for `w_two`.

NEXT UP: CLASS HIERARCHIES

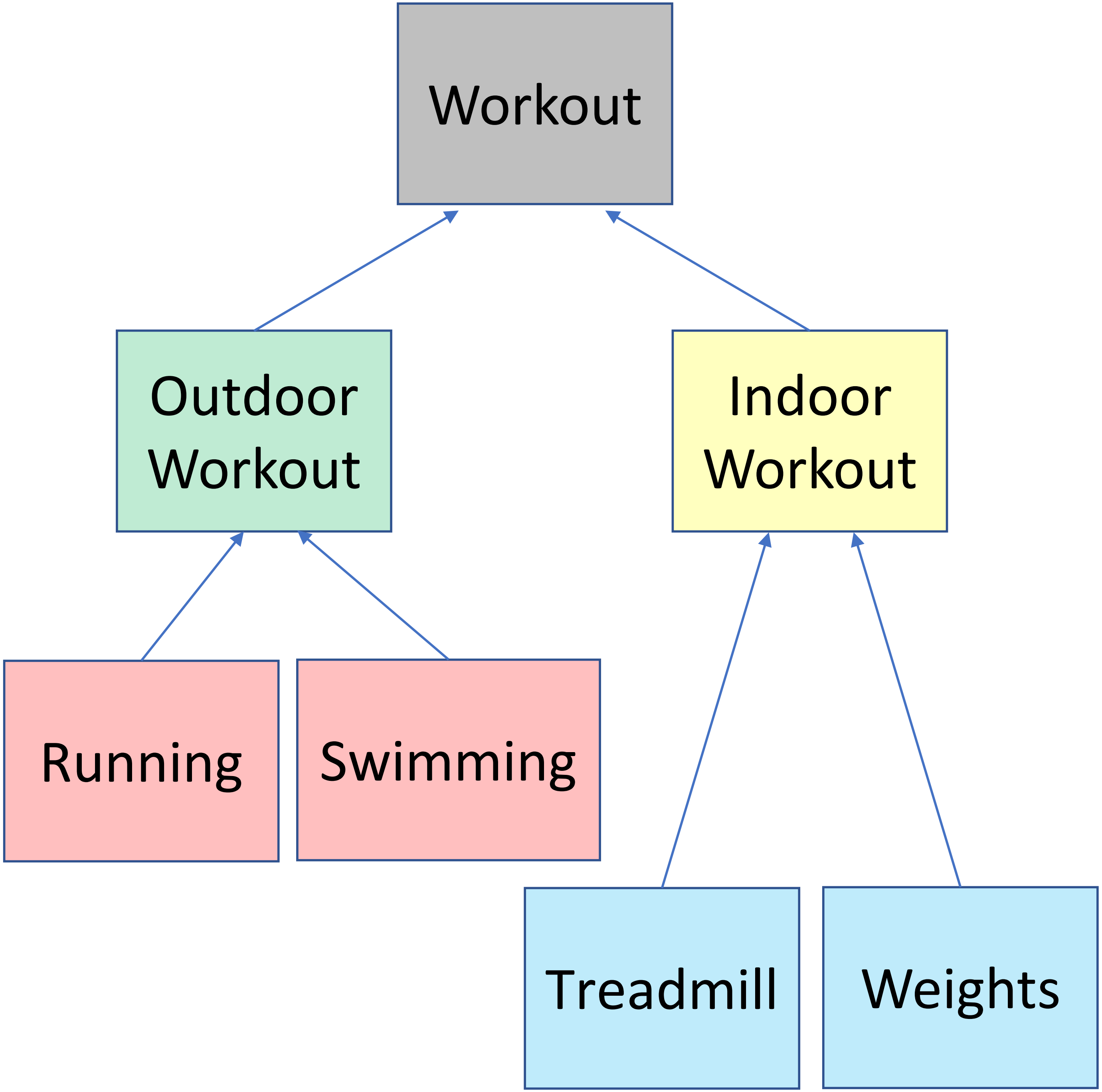




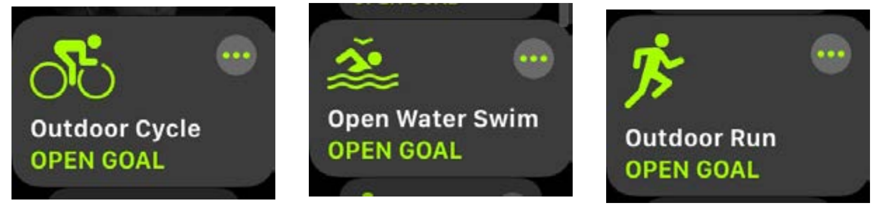


HIERARCHIES

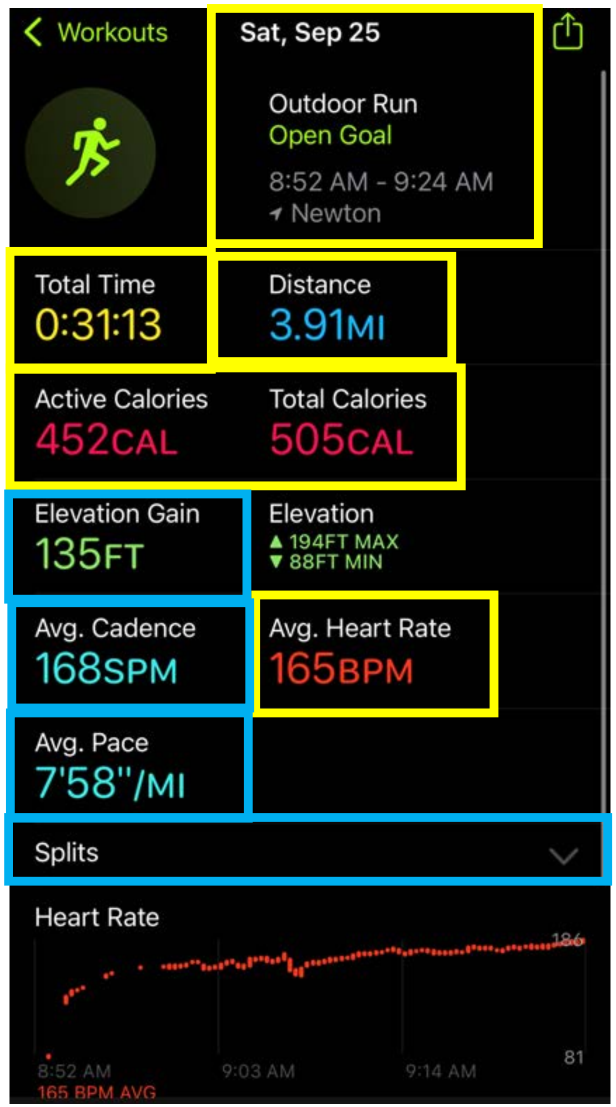
- **Parent class** (superclass)
- **Child class** (subclass)
  - **Inherits** all data and behaviors of parent class
  - **Add** more **info**
  - **Add** more **behavior**
  - **Override** behavior



Fitness Tracker



Different kinds of workouts



Common properties:

Icon | Kind
Date | ...rt
T...
En... | Calories
He...ate | Distance

Workout “Superclass”

Swimming Specific:

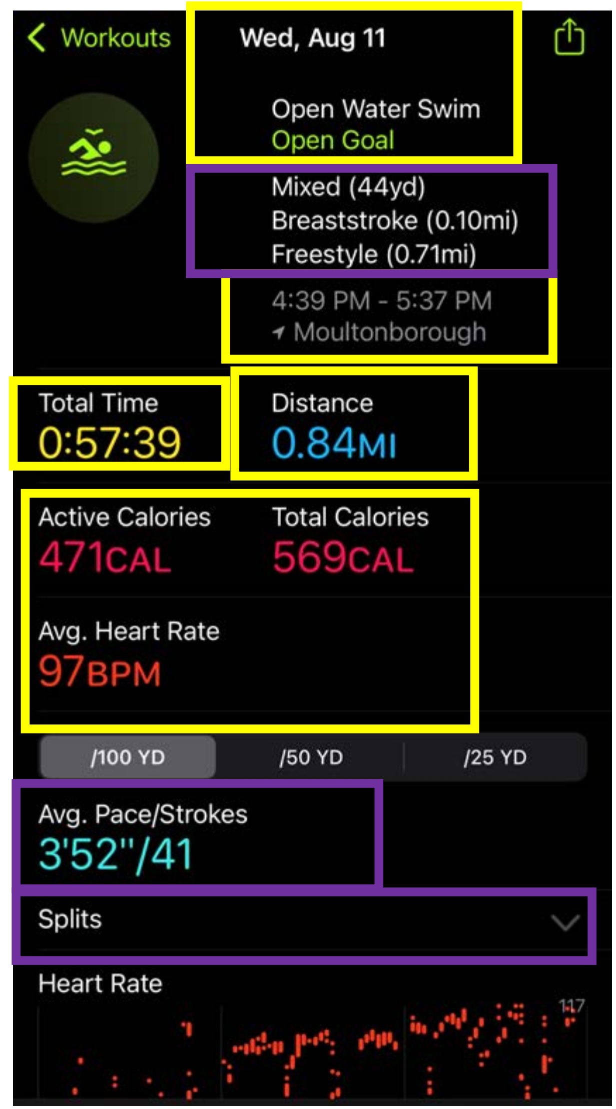
Swimming Pace
Stroke T...
1...

Swimming “Subclass”

Running Specific:

Cadence
Runni...
M...
Ele...

Running “Subclass”



INHERITANCE: PARENT CLASS

```
class Workout(object):
    cal_per_hr = 200
    def __init__(self, start, end, calories=None):
        …
```

- Everything is an object
- Class `object` implements basic operations in Python, e.g., binding variables

INHERITANCE: SUBCLASS

```
class RunWorkout(Workout):

    def __init__(self, start, end, elev=0, calories=None):
        super().__init__(start,end,calories)
        self.icon = '🏃'
        self.kind = 'Running'
        self.elev = elev

    def get_elev(self):
        return self.elev
    def set_elev(self, e):
        self.elev = e
```

Parent is Workout
Inherits all attributes of `Workout`:
start,end,calories
get_calories(), get_start()
get_end(),… ,__str__()

Parent accessed via super()

Add new instance variables

Initialize the parent class (Workout)

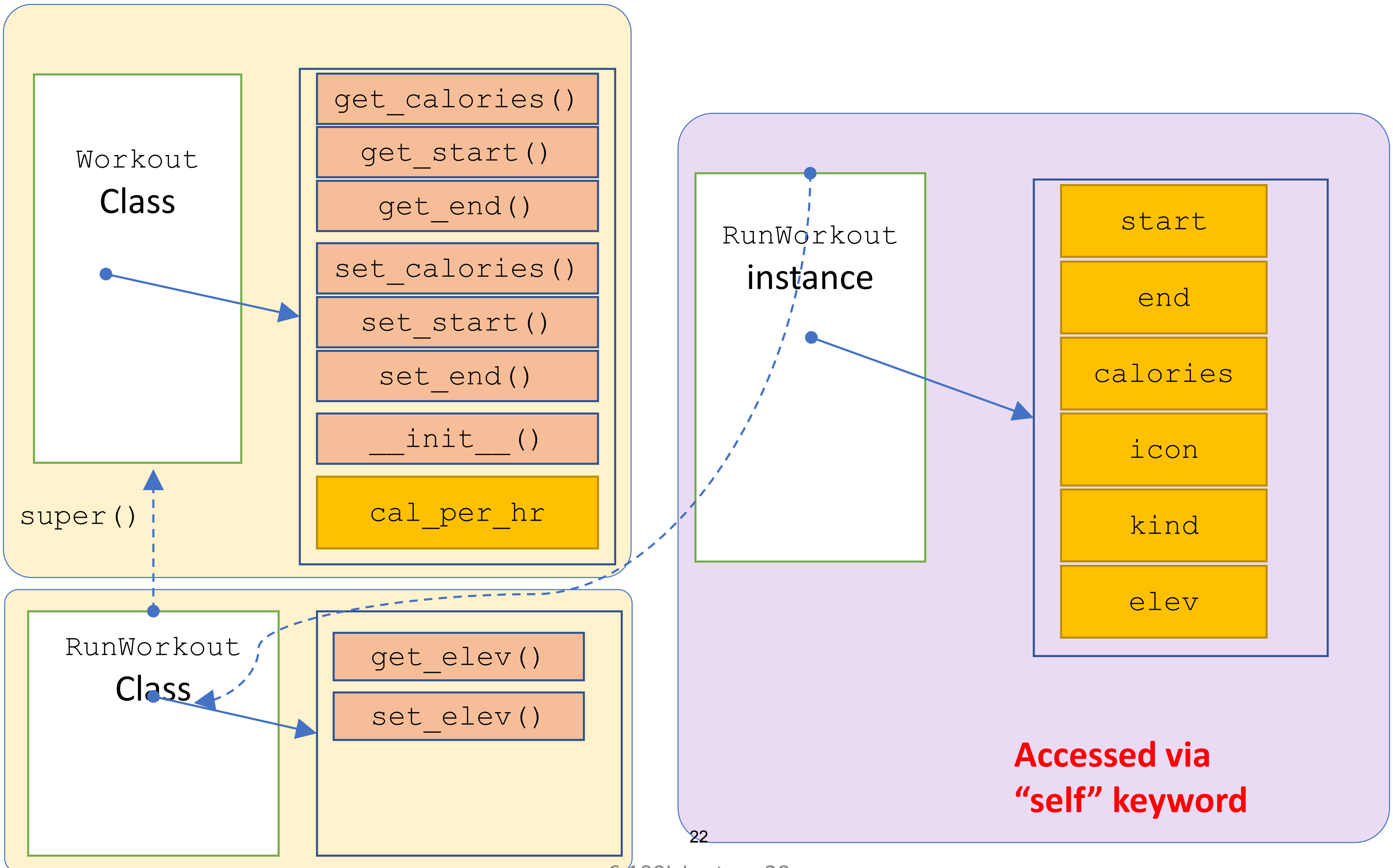
Override parents default

Add new functionality

Add **new functionality** e.g., `get_elev()`

- New methods can be called on instance of type `RunWorkout`
- `__init__` uses `super()` to setup Workout base instance (can also call `Workout.__init__(start,end,calories)` directly

INHERITANCE REPRESENTATION IN MEMORY



WHY USE INHERITENCE?

- Improve **clarity**
  - Commonalities are explicit in parent class
  - Differences are explicit in subclass
- **Reuse** code
- Enhance **modularity**
  - Can pass subclasses to any method that uses parent

SUBCLASSES REUSE PARENT CODE

- Complex print function shared by all subclasses

```
class Workout(object)
.........
    def __str__(self):
        width = 16
        retstr =  f"|{'-'*width}|\n"
        retstr += f"|{' ' *width}|\n"
        iconLen = 0
        retstr += f"| {self.icon}{' '*(width-3)}|\n"
        retstr += f"| {self.kind}{' '*(width-len(self.kind)-1)}|\n"
        retstr += f"|{' ' *width}|\n"
        duration_str = str(self.get_duration())
        retstr += f"| {duration_str}{' '*(width-len(duration_str)-1)}|\n"
        cal_str = f"{self.get_calories():.0f}"
        retstr += f"| {cal_str} Calories {' '*(width-len(cal_str)-11)}|\n"

        retstr += f"|{' ' *width}|\n"
        retstr +=  f"|{'_'*width}|\n"

        return retstr
```

outputs

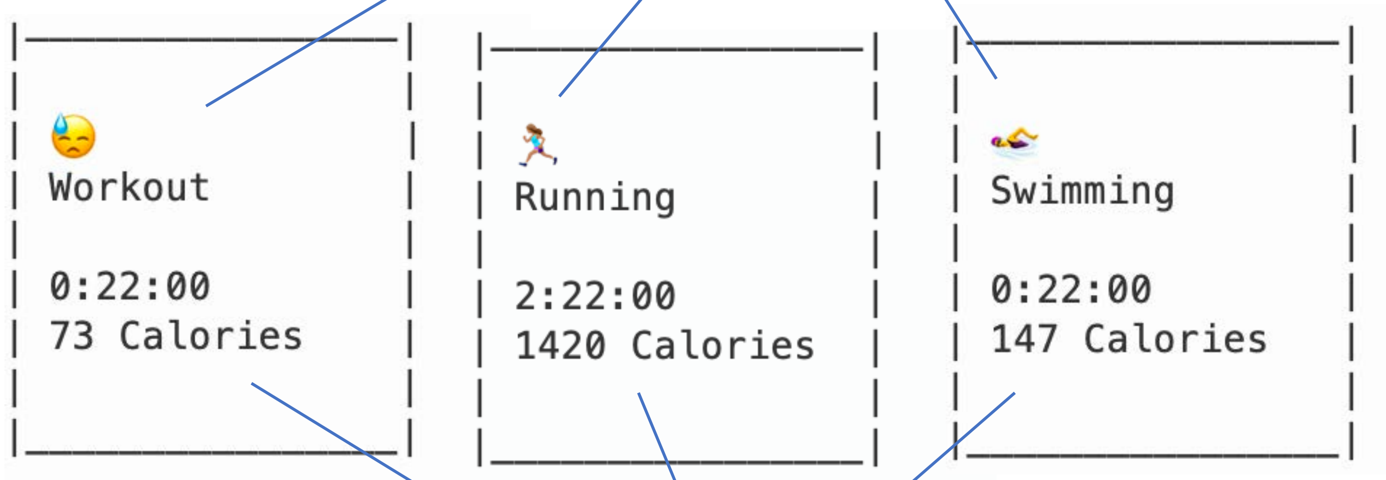
```
|----------------|
|                |
| 😓             |
| Workout        |
|                |
| 0:22:00        |
| 73 Calories    |
|                |
|________________|
```

SUBCLASSES REUSE PARENT CODE

```
w=Workout(…)
rw=RunWorkout(…)
sw=SwimWorkout(…)

print(w)
print(rw)
print(sw)
```

All Workout subclasses can use Workout `__str__()` method!

Workout specific icon and label



Calories calculated based on `cal_per_hr` for each subclass

WHERE CAN I USE AN INSTANCE OF A CLASS?

- We can use an instance of `RunWorkout` anywhere `Workout` can be used
- Opposite is not true (cannot use `Workout` anywhere `RunWorkout` is used)
- Consider two helper functions

```
def total_calories(workouts):
    cals = 0
    for w in workouts:
        cals += w.get_cals()
    return cals
```

```
def total_elevation(run_workouts):
    elev = 0
    for w in run_workouts:
        elev += w.get_elev()
    return elev
```

WHERE CAN I USE AN INSTANCE OF A CLASS?

```
def total_calories(workouts):
    cals = 0
    for w in workouts:
        cals += w.get_cals()
    return cals
```

```
def total_elevation(run_workouts):
    elev = 0
    for w in run_workouts:
        elev += w.get_elev()
    return elev
```

```
w1 = Workout('9/30/2021 1:35 PM','9/30/2021 2:05 PM')
w2 = Workout('9/30/2021 4:35 PM','9/30/2021 5:05 PM')
rw1 = RunWorkout('9/30/2021 1:35 PM','9/30/2021 3:35 PM', 100)
rw2 = RunWorkout('9/30/2021 1:35 PM','9/30/2021 3:35 PM', 200)
```

30 min workouts = 100 cal

2 hr run workouts

elevation val

```
total_calories([w1,w2,rw1,rw2]))   # (1) # cal = 100+100+400+400
total_elevation([rw1,rw2]))        # (2) # elev = 100+200
total_elevation([w1,rw1])          # (3) # err! w1 has no elev method
```

YOU TRY IT!

- For each line creating on object below, tell me:
  - What is the calories val through `get_calories()`
  - What is the elevation val through `get_elev()`

```
w1 = Workout('9/30/2021 2:20 PM','9/30/2021 2:50 PM')
w2 = Workout('9/30/2021 2:20 PM','9/30/2021 2:50 PM',450)
rw1 = RunWorkout('9/30/2021 2:20 PM','9/30/2021 2:50 PM',250)
rw2 = RunWorkout('9/30/2021 2:20 PM','9/30/2021 2:50 PM',250,300)
rw3 = RunWorkout('9/30/2021 2:20 PM','9/30/2021 2:50 PM',calories=300)
```

Demo

OVERRIDING SUPERCLASSES

- Overriding superclass – add calorie calculation w/ distance

```
class RunWorkout(Workout):
    cals_per_km = 100
                ...

    def get_calories(self):
        if (self.route_gps_points != None):
            dist = 0
            lastP = self.routeGpsPoints[0]
            for p in self.routeGpsPoints[1:]:
                dist += gpsDistance(lastP,p)
                lastP = p
            return dist * RunWorkout.cals_per_km
        else:
            return super().get_calories()
```

Add another class var

route_gps_points contains lat/lon pairs of route run

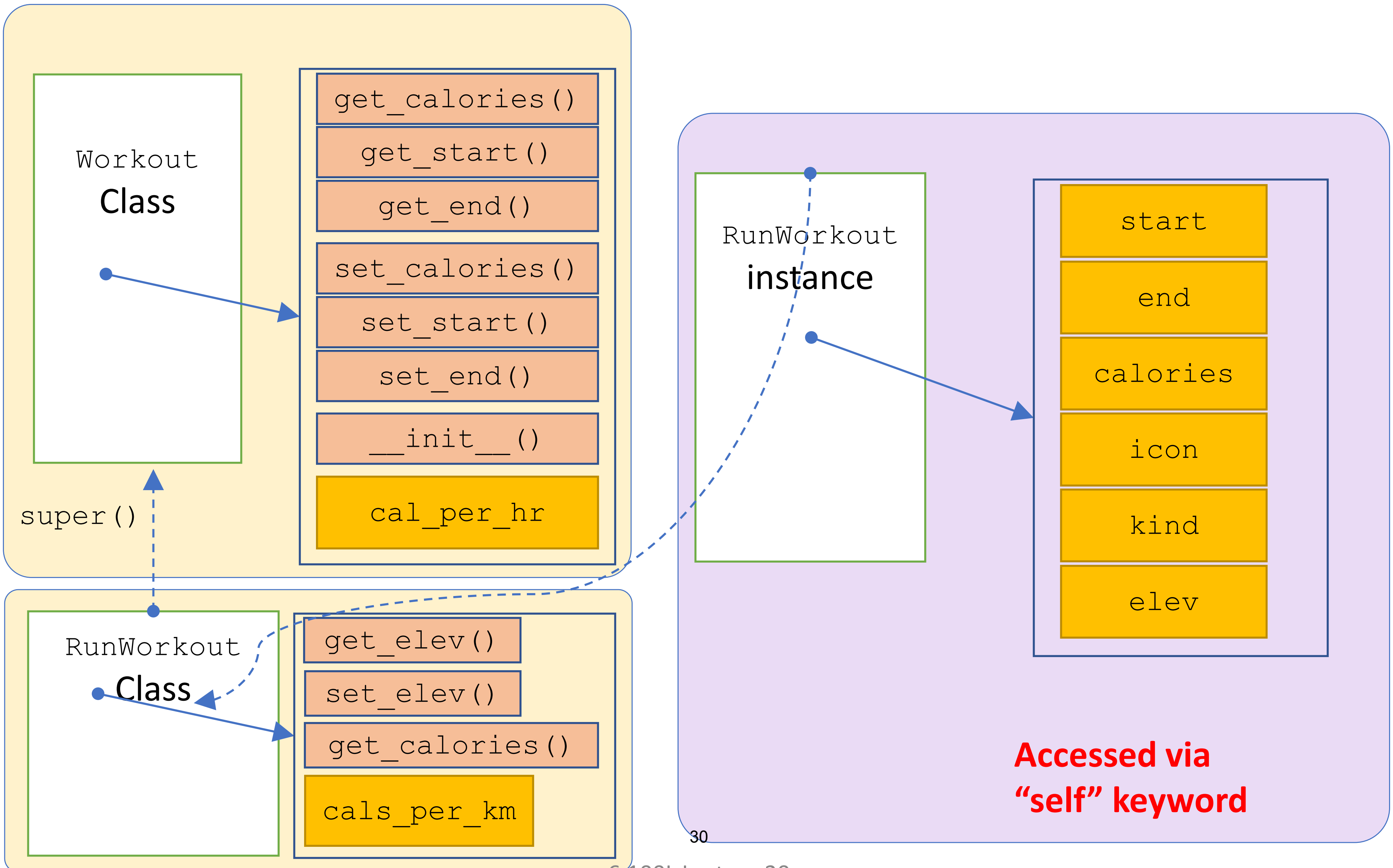
`get_calories()` overridden since it is defined in both sub and superclass

Iterate through all pairs of GPS points

Summing up their distance

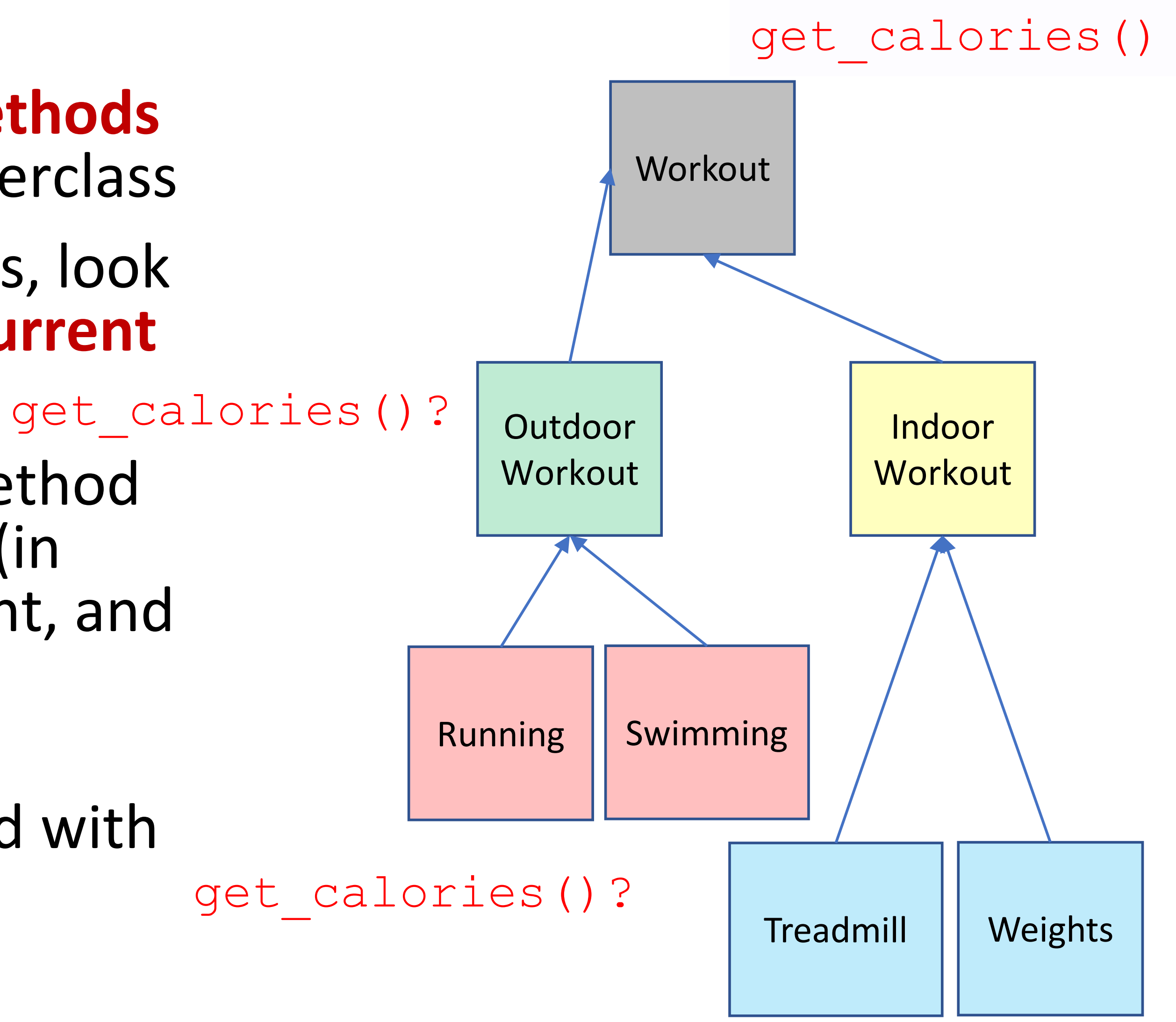
Didn't pass in gps coords, so just do whatever the superclass does

OVERRIDDEN METHODS IN MEMORY



WHICH METHOD WILL BE CALLED?

- **Overriding**: subclass **methods with same name** as superclass
- For an instance of a class, look for a method name in **current class definition**
- If not found, look for method name **up the hierarchy** (in parent, then grandparent, and so on)
- Use first method up the hierarchy that you found with that method name



TESTING EQUALITY WITH SUBCLASSES

- With subclasses, often want to ensure base class is equal, in addition to new properties in the subclass

```
class Workout(object):
......
    def __eq__(self, other):
        return type(self) == type(other) and \
               self.startDate == other.startDate and \
               self.endDate == other.endDate and \
               self.kind == other.kind and \
               self.get_calories() == other.get_calories()
```

Types must be the same

And all the other properties equal too

```
class RunWorkout(Workout):
......
   def __eq__(self,other):
        return super().__eq__(other) and self.elev == other.elev
```

Workout properties are equal

And new properties from RunWorkout are equal

OBJECT ORIENTED DESIGN: MORE ART THAN SCIENCE

- OOP is a powerful tool for **modularizing** your code and grouping state and functions together

BUT

- It's **possible to overdo** it
  - New OOP programmers often create elaborate class hierarchies
  - Not necessarily a good idea
  - Think about the users of your code

    Will your decomposition make sense to them?
  - Because the function that is invoked is implicit in the class hierarchy, it can sometimes be difficult to reason about control flow
- The Internet is full of opinions OOP and "good software design" – you have to **develop your own taste through experience**!

TIMING PROGRAMS, COUNTING OPERATIONS

(download slides and .py files to follow along)

6.100L Lecture 21

Ana Bell

WRITING EFFICIENT PROGRAMS

- So far, we have **emphasized correctness**. It is the first thing to worry about! But sometimes that is not enough.
- Problems can be very complex
- But data sets can be

  very large: in 2014

  Google served

  30,000,000,000,000

  pages covering

  100,000,000 GB

  of data

EFFICIENCY IS IMPORTANT

- Separate **time and space efficiency** of a program
- Tradeoff between them: can use up a bit more memory to store values for quicker lookup later
  - Think Fibonacci recursive vs. Fibonacci with memoization
- Challenges in understanding efficiency
  - A program can be **implemented in many different ways**
  - You can solve a problem using only a handful of different **algorithms**
- Want to separate choice of implementation from choice of more abstract algorithm

EVALUATING PROGRAMS

- Measure with a **timer**
- **Count** the operations
- Abstract notion of **order of growth**

ASIDE on MODULES

- A module is a set of python definitions in a file
  - Python provides many useful modules: math, plotting/graphing, random sampling for probability, statistical tools, many others
- You first need to "import" the module into your environment

```
import time
import random
import dateutil
import math
```

- Call functions from inside the module using the module's name and dot notation

```
math.sin(math.pi/2)
```

TIMING

TIMING A PROGRAM

- Use time module
- Recall that importing means to bring in that class into your own file
- **Start** clock
- **Call** function
- **Stop** clock

```
import time

def c_to_f(c):
    return c*9.0/5 + 32

tstart = time.time()
c_to_f(37)
dt = time.time() - tstart
print(dt, "s,")
```

Seconds since the epoch: Jan 1, 1970

TIMNG `c_to_f`

- Very fast, can’t even time it accurately

```
c_to_f(1) took 0.0 seconds
c_to_f(10) took 0.0 seconds
c_to_f(100) took 0.0 seconds
c_to_f(1000) took 0.0 seconds
c_to_f(10000) took 0.0 seconds
c_to_f(100000) took 0.0 seconds
c_to_f(1000000) took 0.0 seconds
c_to_f(10000000) took 0.0 seconds
```

TIMING `mysum`

- As the **input increases**, the time it takes also increases
- Pattern?
  - 0.009 to 0.05 to 0.5 to 5 to ??

```
mysum(1) took 0.0 sec
mysum(10) took 0.0 sec
mysum(100) took 0.0 sec
mysum(1000) took 0.0 sec
mysum(10000) took 0.0019927024841308594 sec
mysum(100000) took 0.009970903396606445 sec
mysum(1000000) took 0.05089521408081055 sec
mysum(10000000) took 0.49667453765869 14 sec
mysum(100000000) took 5.688449382781982 sec
```

TIMING `square`

- As the **input increases** the time it takes also increases
- `square` called with 100000 did not finish within a reasonable amount of time
- Maybe we can guess a pattern if we are patient for one more round?

```
square(1) took 0.0 sec
square(10) took 0.0 sec
square(100) took 0.0 sec
square(1000) took 0.06244492530822754 sec
square(10000) took 5.553335428237915 sec
```

TIMING PROGRAMS IS INCONSISTENT

- GOAL: to evaluate different algorithms
- Running time **should vary between algorithms**
- Running time **should not vary between implementations**
- Running time **should not vary between computers**
- Running time **should not vary between languages**
- Running time is **should be predictable** for small inputs

- Time varies for different inputs but cannot really express a relationship between inputs and time needed
- Can only be measured *a posteriori*

COUNTING

COUNTING OPERATIONS

- Assume these steps take **constant time**:
  - Mathematical operations
  - Comparisons
  - Assignments
  - Accessing objects in memory
- Count number of operations executed as function of size of input

c_to_f → 3 ops

```
def c_to_f(c):
    return c*9.0/5 + 32
```

3 ops

mysum → 1+(x+1)*(1+2) = 3x+4 ops

```
def mysum(x):
    total = 0
    for i in range(x+1):
        total += i
    return total
```

1 op

loop x+1 times

1 op

2 ops

square →1+n*(1)*n*(1+2) = 3n² + 1 ops

```
def square(n):
    sqsum = 0
    for i in range(n):
        for j in range(n):
            sqsum += 1
    return sqsum
```

1 op

1 op

1 op

loop n times

loop n times

2 ops

COUNTING `c_to_f`

- No matter what the input is, the number of **operations is the same**

```
c_to_f(100): 3 ops, 1.0 x more
c_to_f(1000): 3 ops, 1.0 x more
c_to_f(10000): 3 ops, 1.0 x more
c_to_f(100000): 3 ops, 1.0 x more
c_to_f(1000000): 3 ops, 1.0 x more
c_to_f(10000000): 3 ops, 1.0 x more
```

COUNTING `mysum`

- As the input increases **by 10**, the number if operations ran is approx. **10 times more**.

```
mysum(100): 304 ops, 1.0 x more
mysum(1000): 3004 ops, 9.88158 x more
mysum(10000): 30004 ops, 9.98802 x more
mysum(100000): 300004 ops, 9.9988 x more
mysum(1000000): 3000004 ops, 9.99988 x more
mysum(10000000): 30000004 ops, 9.99999 x more
```

COUNTING `square`

- As the input increases **by 10**, the number of operations is approx. **100 times more**.

```
square(1): 5 ops, 1.0 x more
square(10): 311 ops, 62.2 x more
square(100): 30101 ops, 96.78778 x more
square(1000): 3001001 ops, 99.69772 x more
square(10000): 300010001 ops, 99.96998 x more
```

- As the input increases **by 2**, the number of operations is approx. **4 times more**.

```
square(128): 49281 ops, 1.0 x more
square(256): 196865 ops, 3.99474 x more
square(512): 786945 ops, 3.99738 x more
square(1024): 3146753 ops, 3.99869 x more
square(2048): 12584961 ops, 3.99935 x more
square(4096): 50335745 ops, 3.99967 x more
square(8192): 201334785 ops, 3.99984 x more
```

COUNTING OPERATIONS IS INDEPENDENT OF COMPUTER VARIATIONS, BUT …

- GOAL: to evaluate different algorithms
- Running “time” **should vary between algorithms**
- Running “time” **should not vary between implementations**
- Running “time” **should not vary between computers**
- Running “time” **should not vary between languages**
- Running “time” is **should be predictable** for small inputs
- No real definition of **which operations** to count

- Count varies for different inputs and can derive a relationship between inputs and the count

… STILL NEED A BETTER WAY

- Timing and counting **evaluate implementations**
- Timing and counting **evaluate machines**

- Want to **evaluate algorithm**
- Want to **evaluate scalability**
- Want to **evaluate in terms of input size**

BIG OH and THETA

(download slides and .py files to follow along)

6.100L Lecture 22

Ana Bell

TIMING

TIMING A PROGRAM

- Use time module
- Importing means bringing collection of functions into your own file
- **Start** clock
- **Call** function
- **Stop** clock

```
import time

def convert_to_km(m):
    return m * 1.609
t0 = time.perf_counter()
convert_to_km(100000)
dt = time.perf_counter() - t0
print("t =", dt, "s,")
```

More accurate timer, meaningful when used to get a time diff

EXAMPLE: convert_to_km, compound

```
def convert_to_km(m):
    return m * 1.609

def compound(invest, interest, n_months):
    total=0
    for i in range(n_months):
        total = total * interest + invest
    return total
```

- How long does it take to compute these functions?
- Does the time depend on the input parameters?
- Are the times noticeably different for these two functions?

CREATING AN INPUT LIST

Create a set of input sizes,
each of which is 10 times
larger than the previous one
[1, 10, 100, 1000, ...]

```
L_N = [1]
for i in range(7):
    L_N.append(L_N[-1]*10)

for N in L_N:
    t = time.perf_counter()
    km = convert_to_km(N)
    dt = time.perf_counter()-t
    print(f"convert_to_km({N}) took {dt} seconds ({1/dt}/sec)")
```

Measure time to compute (aka
run function) for each input

Report time and how many
times the fcn can run per sec

RUN IT!
`convert_to_km` OBSERVATIONS

Scientific notation, i.e.

$1.44e\text{–}06 = 1.44 \times 10^{-6}$

```
convert_to_km(1) took 4.30e-06 sec (232,558.14/sec)
convert_to_km(10) took 7.00e-07 sec (1,428,571.43/sec)
convert_to_km(100) took 4.00e-07 sec (2,499,999.99/sec)
convert_to_km(1000) took 3.00e-07 sec (3,333,333.33/sec)
convert_to_km(10000) took 3.00e-07 sec (3,333,333.33/sec)
convert_to_km(100000) took 4.00e-07 sec (2,499,999.99/sec)
convert_to_km(1000000) took 4.00e-07 sec (2,499,999.99/sec)
convert_to_km(10000000) took 3.00e-07 sec (3,333,333.33/sec)
convert_to_km(100000000) took 3.00e-07 sec (3,333,333.33/sec)
```

Observation: average time seems independent of size of argument

MEASURE TIME:
compound with a variable number of months

```
def compound(invest, interest, n_months):
    total=0
    for i in range(n_months):
        total = total * interest + invest
    return total
```

compound(1) took 2.26e-06 seconds (441,696.12/sec)
compound(10) took 2.31e-06 seconds (433,839.48/sec)
compound(100) took 6.59e-06 seconds (151,676.02/sec)
compound(1000) took 5.02e-05 seconds (19,938.59/sec)
compound(10000) took 5.10e-04 seconds (1,961.80/sec)
compound(100000) took 5.14e-03 seconds (194.46/sec)
compound(1000000) took 4.79e-02 seconds (20.86/sec)
compound(10000000) took 4.46e-01 seconds (2.24/sec)

Observation 1: Time grows with the input only when n_months changes

Observation 2: average time seems to increase by 10 as size of argument increases by 10

Observation 3: relationship between size and time only predictable for large sizes

MEASURE TIME: sum over L

```
def sum_of(L):
    total = 0.0
    for elt in L:
        total = total + elt
    return total

L_N = [1]
for i in range(7):
    L_N.append(L_N[-1]*10)

for N in L_N:
    L = list(range(N))
    t = time.perf_counter()
    s = sum_of(L)
    dt = time.perf_counter()-t
    print(f"sum_of({N}) took {dt} seconds ({1/dt}/sec)")
```

[0,1,2,...9] then
[0,1,2,...99] etc

Observation 1: Size of the input is now the length of the list, not how big the element numbers are.

Observation 2: average time seems to increase by 10 as size of argument increases by 10

Observation 3: relationship between size and time only predictable for large sizes

Observation 4: Time seems comparable to computation of compound

MEASURE TIME: find element in a list

```
# search each element one-by-one
def is_in(L, x):
    for elt in L:
        if elt==x:
            return True
    return False

# search by bisecting the list (list should be sorted!)
def binary_search(L, x):
    lo = 0
    hi = len(L)
    while hi-lo > 1:
        mid = (hi+lo) // 2
        if L[mid] <= x:
            lo = mid
        else:
            hi = mid
    return L[lo] == x

# search using built-in operator
x in L
```

Integer division, round down

Measure "average" time. Search for the first, middle, and last element of sorted list, and average these 3 times.

MEASURE TIME: find element in a list

```
is_in(10000000) took 1.62e-01 seconds (6.16/sec)
    9.57 times more than for 10 times fewer elements
binary(10000000) took 9.37e-06 seconds (106,761.64/sec)
    1.40 times more than for 10 times fewer elements
builtin(10000000) took 5.64e-02 seconds (17.72/sec)
    9.63 times more than for 10 times fewer elements

is_in(100000000) took 1.64e+00 seconds (0.61/sec)
    10.12 times more than for 10 times fewer elements
binary(100000000) took 1.18e-05 seconds (84,507.09/sec)
    1.26 times more than for 10 times fewer elements
builtin(100000000) took 5.70e-01 seconds (1.75/sec)
    10.11 times more than for 10 times fewer elements
```

Observation 1: searching one-by-one grows by factor of 10, when L increases by 10

MEASURE TIME: find element in a list

```
is_in(10000000) took 1.62e-01 seconds (6.16/sec)
    9.57 times more than for 10 times fewer elements
binary(10000000) took 9.37e-06 seconds (106,761.64/sec)
    1.40 times more than for 10 times fewer elements
builtin(10000000) took 5.64e-02 seconds (17.72/sec)
    9.63 times more than for 10 times fewer elements

is_in(100000000) took 1.64e+00 seconds (0.61/sec)
    10.12 times more than for 10 times fewer elements
binary(100000000) took 1.18e-05 seconds (84,507.09/sec)
    1.26 times more than for 10 times fewer elements
builtin(100000000) took 5.70e-01 seconds (1.75/sec)
    10.11 times more than for 10 times fewer elements
```

Observation 1: searching one-by-one grows by factor of 10, when L increases by 10

Observation 2: built-in function grows by factor of 10, when L increases by 10

MEASURE TIME: find element in a list

```
is_in(10000000) took 1.62e-01 seconds (6.16/sec)
    9.57 times more than for 10 times fewer elements
binary(10000000) took 9.37e-06 seconds (106,761.64/sec)
    1.40 times more than for 10 times fewer elements
builtin(10000000) took 5.64e-02 seconds (17.72/sec)
    9.63 times more than for 10 times fewer elements

is_in(100000000) took 1.64e+00 seconds (0.61/sec)
    10.12 times more than for 10 times fewer elements
binary(100000000) took 1.18e-05 seconds (84,507.09/sec)
    1.26 times more than for 10 times fewer elements
builtin(100000000) took 5.70e-01 seconds (1.75/sec)
    10.11 times more than for 10 times fewer elements
```

Observation 1: searching one-by-one grows by factor of 10, when L increases by 10

Observation 2: built-in function grows by factor of 10, when L increases by 10

Observation 3: binary search time seems *almost* independent of size

MEASURE TIME: find element in a list

```
is_in(10000000) took 1.62e-01 seconds (6.16/sec)
    9.57 times more than for 10 times fewer elements
binary(10000000) took 9.37e-06 seconds (106,761.64/sec)
    1.40 times more than for 10 times fewer elements
builtin(10000000) took 5.64e-02 seconds (17.72/sec)
    9.63 times more than for 10 times fewer elements

is_in(100000000) took 1.64e+00 seconds (0.61/sec)
    10.12 times more than for 10 times fewer elements
binary(100000000) took 1.18e-05 seconds (84,507.09/sec)
    1.26 times more than for 10 times fewer elements
builtin(100000000) took 5.70e-01 seconds (1.75/sec)
    10.11 times more than for 10 times fewer elements
```

Observation 1: searching one-by-one grows by factor of 10, when L increases by 10

Observation 2: built-in function grows by factor of 10, when L increases by 10

Observation 3: binary search time seems *almost* independent of size

Observation 4: binary search much faster than is_in, especially on larger problems

MEASURE TIME: find element in a list

```
is_in(10000000) took 1.62e-01 seconds (6.16/sec)
    9.57 times more than for 10 times fewer elements
binary(10000000) took 9.37e-06 seconds (106,761.64/sec)
    1.40 times more than for 10 times fewer elements
builtin(10000000) took 5.64e-02 seconds (17.72/sec)
    9.63 times more than for 10 times fewer elements

is_in(100000000) took 1.64e+00 seconds (0.61/sec)
    10.12 times more than for 10 times fewer elements
binary(100000000) took 1.18e-05 seconds (84,507.09/sec)
    1.26 times more than for 10 times fewer elements
builtin(100000000) took 5.70e-01 seconds (1.75/sec)
    10.11 times more than for 10 times fewer elements
```

Observation 1: searching one-by-one grows by factor of 10, when L increases by 10

Observation 2: built-in function grows by factor of 10, when L increases by 10

Observation 3: binary search time seems *almost* independent of size

Observation 4: binary search much faster than is_in, especially on larger problems

Observation 5: is_in is slightly slower than using Python's "in" capability

MEASURE TIME: find element in a list

```
def is_in(L, x):
    for elt in L:
        if elt==x:
            return True
    return False


def binary_search(L, x):
    lo = 0
    hi = len(L)
    while hi-lo > 1:
        mid = (hi+lo) // 2
        if L[mid] <= x:
            lo = mid
        else:
            hi = mid
    return L[lo] == x
```

So we have seen computations where time seems very different

- Constant time
- Linear in size of argument
- Something less than linear?

MEASURE TIME: diameter function

```
L=[(cos(0), sin(0)),
   (cos(1), sin(1)),
   (cos(2), sin(2)), ... ] #example numbers

def diameter(L):
    farthest_dist = 0
    for i in range(len(L)):
        for j in range(i+1, len(L)):
            p1 = L[i]
            p2 = L[j]
            dist = math.sqrt((p1[0]-p2[0])**2+(p1[1]-p2[1])**2)
            if dist > farthest_dist:
                farthest_dist = dist
    return farthest_dist
```

1st iter: len(L) - 1 passes
2nd iter: len(L) - 2 pases ...
On average, len(L) / 2 passes

L = [(cos(0),sin(0)), (cos(1),sin(1)), (cos(2),sin(2)), (cos(3),sin(3))]

MEASURE TIME: diameter function

```
def diameter(L):
    farthest_dist = 0
    for i in range(len(L)):
        for j in range(i+1, len(L)):
            p1 = L[i]
            p2 = L[j]
            dist = math.sqrt((p1[0]-p2[0])**2+(p1[1]-p2[1])**2)
            if dist > farthest_dist:
                farthest_dist = dist
    return farthest_dist
```

1st iter: len(L) - 1 passes
2nd iter: len(L) - 2 pases ...
On average, len(L) / 2 passes

`L = [(cos(0),sin(0)), (cos(1),sin(1)), (cos(2),sin(2)), (cos(3),sin(3))]`

MEASURE TIME: diameter function

```
def diameter(L):
    farthest_dist = 0
    for i in range(len(L)):
        for j in range(i+1, len(L)):
            p1 = L[i]
            p2 = L[j]
            dist = math.sqrt((p1[0]-p2[0])**2+(p1[1]-p2[1])**2)
            if dist > farthest_dist:
                farthest_dist = dist
    return farthest_dist
```

1st iter: len(L) - 1 passes
2nd iter: len(L) - 2 pases …
On average, len(L) / 2 passes

L = [(cos(0),sin(0)), (cos(1),sin(1)), (cos(2),sin(2)), (cos(3),sin(3))]

MEASURE TIME: diameter function

```
def diameter(L):
    farthest_dist = 0
    for i in range(len(L)):
        for j in range(i+1, len(L)):
            p1 = L[i]
            p2 = L[j]
            dist = math.sqrt((p1[0]-p2[0])**2+(p1[1]-p2[1])**2)
            if dist > farthest_dist:
                farthest_dist = dist
    return farthest_dist
```

1st iter: len(L) - 1 passes
2nd iter: len(L) - 2 pases ...
On average, len(L) / 2 passes

```
L = [(cos(0),sin(0)), (cos(1),sin(1)), (cos(2),sin(2)), (cos(3),sin(3))]
```

MEASURE TIME: diameter function

```
def diameter(L):
    farthest_dist = 0
    for i in range(len(L)):
        for j in range(i+1, len(L)):
            p1 = L[i]
            p2 = L[j]
            dist = math.sqrt((p1[0]-p2[0])**2+(p1[1]-p2[1])**2)
            if dist > farthest_dist:
                farthest_dist = dist
    return farthest_dist
```

1st iter: len(L) - 1 passes
2nd iter: len(L) - 2 pases ...
On average, len(L) / 2 passes

```
L = [(cos(0),sin(0)), (cos(1),sin(1)), (cos(2),sin(2)), (cos(3),sin(3))]
```

MEASURE TIME: diameter function

```
def diameter(L):
    farthest_dist = 0
    for i in range(len(L)):
        for j in range(i+1, len(L)):
            p1 = L[i]
            p2 = L[j]
            dist = math.sqrt((p1[0]-p2[0])**2+(p1[1]-p2[1])**2)
            if dist > farthest_dist:
                farthest_dist = dist
    return farthest_dist
```

1st iter: len(L) - 1 passes
2nd iter: len(L) - 2 pases …
On average, len(L) / 2 passes

```
L = [(cos(0),sin(0)), (cos(1),sin(1)), (cos(2),sin(2)), (cos(3),sin(3))]
```

MEASURE TIME: diameter function

```
def diameter(L):
    farthest_dist = 0
    for i in range(len(L)):
        for j in range(i+1, len(L)):
            p1 = L[i]
            p2 = L[j]
            dist = math.sqrt((p1[0]-p2[0])**2+(p1[1]-p2[1])**2)
            if dist > farthest_dist:
                farthest_dist = dist
    return farthest_dist
```

1st iter: len(L) - 1 passes
2nd iter: len(L) - 2 pases ...
On average, len(L) / 2 passes

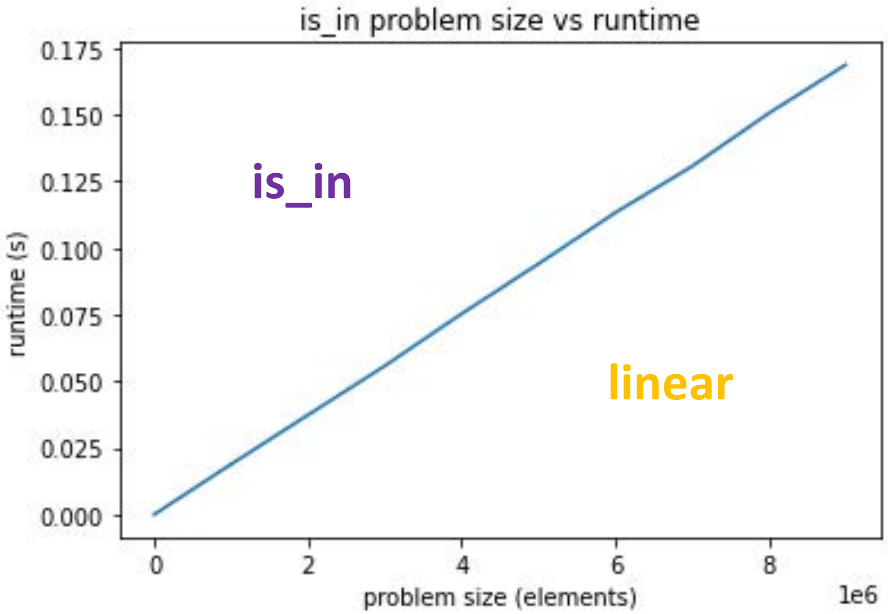
```
L = [(cos(0),sin(0)), (cos(1),sin(1)), (cos(2),sin(2)), (cos(3),sin(3))]
```

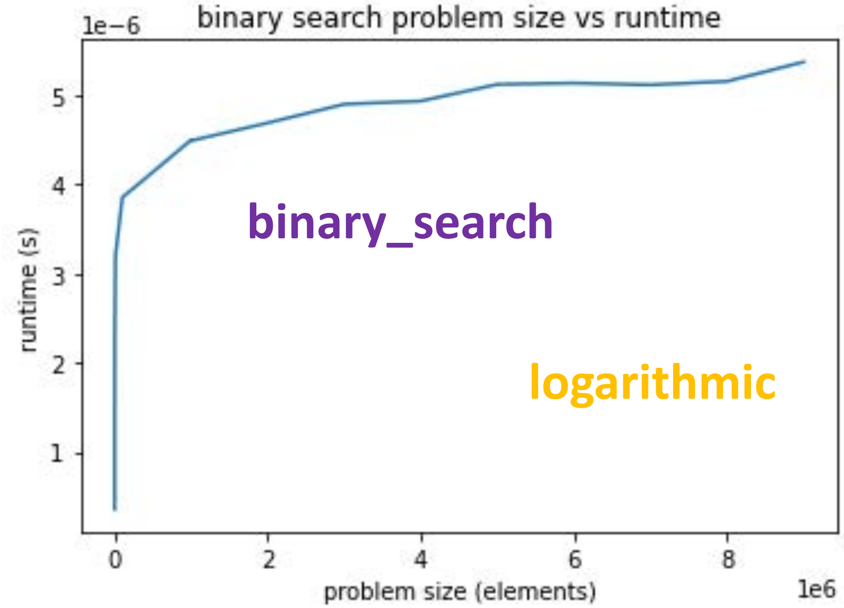
MEASURE TIME: diameter function

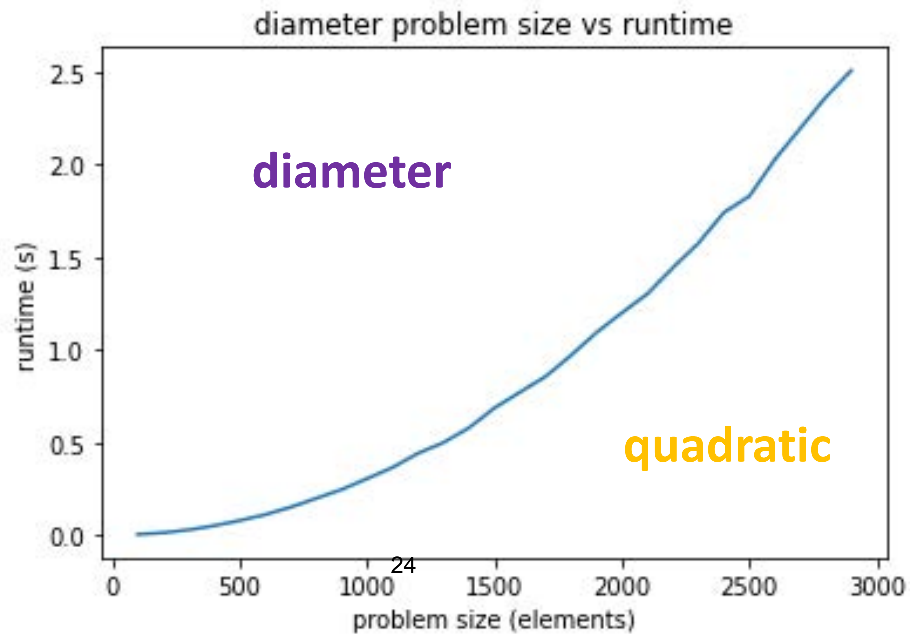
```
def diameter(L):
    farthest_dist = 0
    for i in range(len(L)):
        for j in range(i+1, len(L)):
            p1 = L[i]
            p2 = L[j]
            dist = math.sqrt((p1[0]-p2[0])**2+(p1[1]-p2[1])**2)
            if dist > farthest_dist:
                farthest_dist = dist
    return farthest_dist
```

- Gets much slower as size of input grows
- *Quadratic: for list of size len(L), does len(L)/2 operations per element on average*
- *len(L) x len(L)/2 operations — worse than linear growth*

PLOT OF INPUT SIZE vs. TIME TO RUN







TWO DIFFERENT MACHINES

My old laptop

```
convert( 1 ) took  0.0919969081879 seconds
convert( 10 ) took  0.0812351703644 seconds
convert( 100 ) took  0.0810060501099 seconds
convert( 1000 ) took  0.0786969661713 seconds
convert( 10000 ) took  0.0776309967041 seconds
convert( 100000 ) took  0.0800149440765 seconds
convert( 1000000 ) took  0.0772659778595 seconds
convert( 10000000 ) took  0.0839469432831 seconds
convert( 100000000 ) took  0.0802690982819 seconds
convert( 1000000000 ) took  0.0796220302582 seconds
compound( 1 ) took  0.0781879425049 seconds
compound( 10 ) took  0.0791871547699 seconds
compound( 100 ) took  0.0802779197693 seconds
compound( 1000 ) took  0.0811159610748 seconds
compound( 10000 ) took  0.079794883728 seconds
compound( 100000 ) took  0.0803499221802 seconds
compound( 1000000 ) took  0.180749893188 seconds
compound( 10000000 ) took  0.713826179504 seconds
compound( 100000000 ) took  6.48052787781 seconds
compound( 1000000000 ) took  63.5682651997 seconds
```

My old desktop

```
convert( 1 ) took  0.0651700496674 seconds
convert( 10 ) took  0.0838208198547 seconds
convert( 100 ) took  0.0830719470978 seconds
convert( 1000 ) took  0.0816540718079 seconds
convert( 10000 ) took  0.0824558734894 seconds
convert( 100000 ) took  0.0837979316711 seconds
convert( 1000000 ) took  0.0837349891663 seconds
convert( 10000000 ) took  0.0843281745911 seconds
convert( 100000000 ) took  0.0838270187378 seconds
convert( 1000000000 ) took  0.0844709873199 seconds
compound( 1 ) took  0.08348703384 seconds
compound( 10 ) took  0.0834701061249 seconds
compound( 100 ) took  0.08316302295 seconds
compound( 1000 ) took  0.0843181610107 seconds
compound( 10000 ) took  0.0845410823822 seconds
compound( 100000 ) took  0.099858045578 seconds
compound( 1000000 ) took  0.183917045593 seconds
compound( 10000000 ) took  1.38667988777 seconds
compound( 100000000 ) took  12.7653880119 seconds
compound( 1000000000 ) took  126.978576899 seconds
```

~2x slower for large problems

Observation 1: even for the same code, the actual machine may affect speed.

Observation 2: Looking only at the relative increase in run time from a prev run, if input is n times as big, the run time is approx. n times as long.

DON'T GET ME WRONG!

- Timing is a **critical tool to assess the performance** of programs
  - At the end of the day, it is irreplaceable for real-world assessment
- But we will see a complementary tool (**asymptotic complexity**) that has other advantages
  - A priori evaluation (before writing or running code)
  - **Assesses algorithm** independent of machine and implementation (what is intrinsic efficiency of algorithm?)
  - Provides direct insight into the **design** of efficient algorithms

COUNTING

COUNT OPERATIONS

- Assume these steps take **constant time**:
  - Mathematical operations
  - Comparisons
  - Assignments
  - Accessing objects in memory
- Count number of these operations executed as function of size of input

convert_to_km → 2 ops

```
def convert_to_km(m):
    return m * 1.609
```

2 ops

sum_of → 1+len(L)*3+1 = 3*len(L)+2 ops

```
def sum_of(L):
    total = 0
    for i in L:
        total += i
    return total
```

1 op

1 op

loop len(L) times

2 ops

1 op

COUNT OPERATIONS: is_in

```
def is_in_counter(L, x):

    for elt in L:

        if elt==x:
            return True
    return False
```

COUNT OPERATIONS: is_in

```
def is_in_counter(L, x):
    global count
    count += 1
    for elt in L:
        count += 2
        if elt==x:
            return True
    return False
```

Return value

Set elt as val from L,
Check elt==x

Global lets us reference and change an external variable inside a function – OK for debugging / timing but not good practice in real programs

COUNT OPERATIONS: binary search

```
def binary_search_counter(L, x):
    global count
    lo = 0
    hi = len(L)
    count += 3
    while hi-lo > 1:
        count += 2
        mid = (hi+lo) // 2
        count += 3
        if L[mid] <= x:
            lo = mid
        else:
            hi = mid
        count += 3
    count += 3
    return L[lo] == x
```

Set lo, hi, len

While test and the subtraction

Addition, //, and assign mid

Access mid, if test and assign mid

Access lo, == test, return

COUNT OPERATIONS

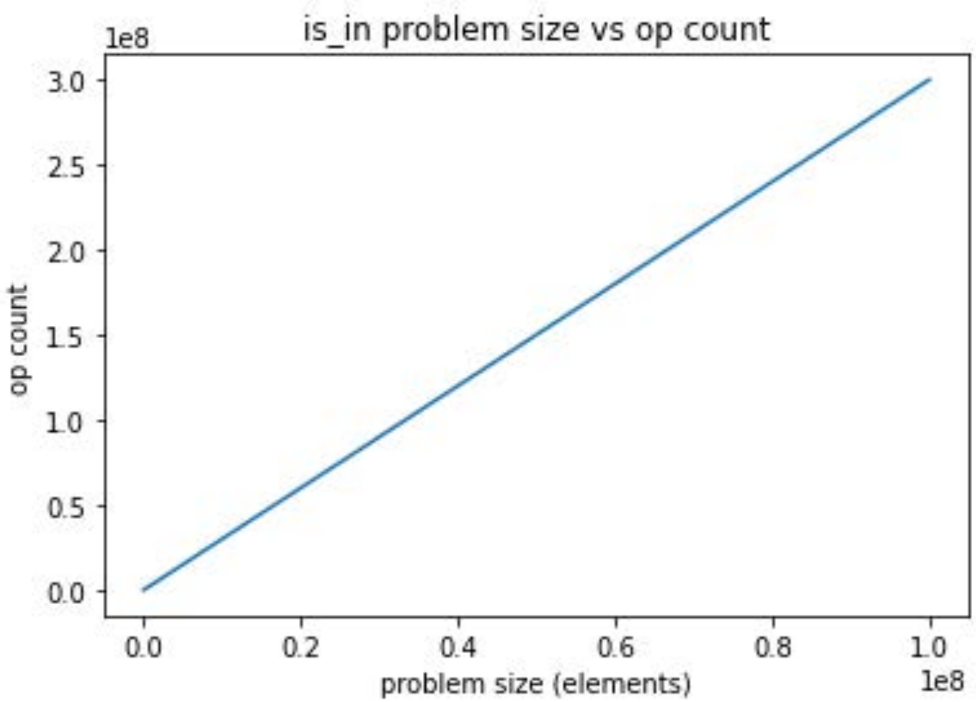
is_in testing
for 1 element, is_in used 9 operations
for 10 element, is_in used 37 operations
for 100 element, is_in used 307 operations
for 1000 element, is_in used 3007 operations
for 10000 element, is_in used 30007 operations
for 100000 element, is_in used 300007 operations
for 1000000 element, is_in used 3000007 operations

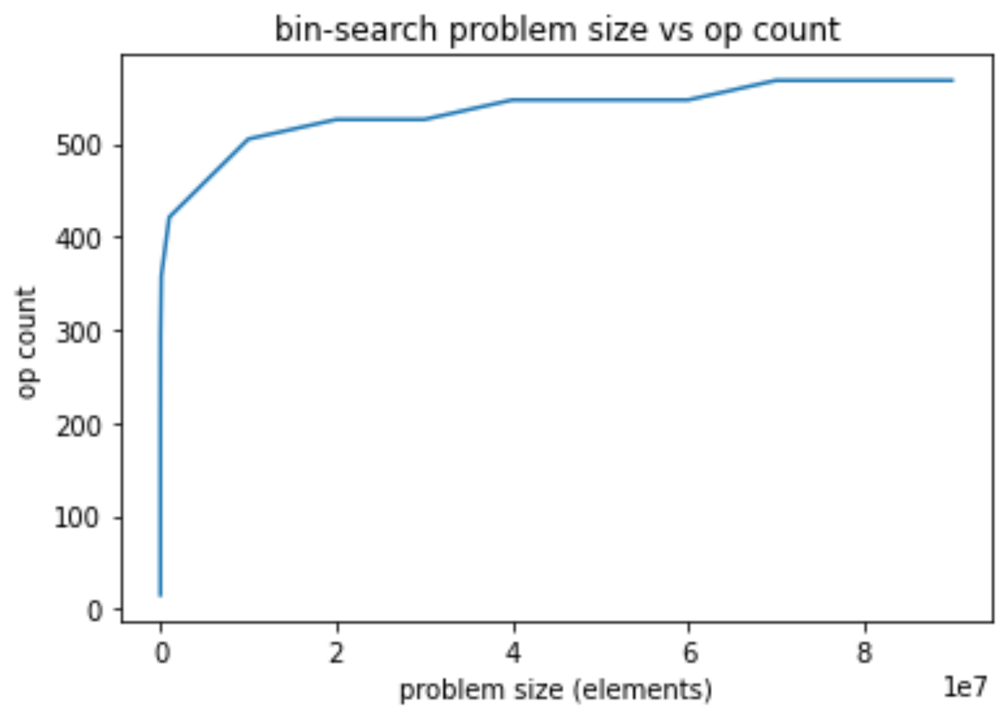
Observation 1: number of operations for is_in increases by 10 as size increases by 10

binary_search testing
for 1 element, binary search used 15 operations
for 10 element, binary search used 85 operations
for 100 element, binary search used 148 operations
for 1000 element, binary search used 211 operations
for 10000 element, binary search used 295 operations
for 100000 element, binary search used 358 operations
for 1000000 element, binary search used 421 operations

Observation 2: *but* number of operations for binary search grows *much more slowly.* Unclear at what rate.

PLOT OF INPUT SIZE vs. OPERATION COUNT





PROBLEMS WITH TIMING AND COUNTING

- **Timing** the exact running time of the program
  - Depends on **machine**
  - Depends on **implementation**
  - **Small inputs** don't show growth
- **Counting** the exact number of steps
  - Gets us a **formula!**
  - **Machine independent**, which is good
  - Depends on **implementation**
  - **Multiplicative/additive constants** are irrelevant for large inputs
- Want to:
  - evaluate **algorithm**
  - evaluate **scalability**
  - evaluate **in terms of input size**

EFFICIENCY IN TERMS OF INPUT: BIG-PICTURE

RECALL `mysum` (one loop) and `square` (nested loops)

- `mysum(x)`
  - What happened to the **program efficiency as x increased**?
  - 10 times bigger x meant the program
    - Took approx. 10 times as long to run
    - Did approx. 10 times as many ops
  - Express it in an “order of” way vs. the input variable: **efficiency = Order of x**
- `square(x)`
  - What happened to the **program efficiency as x increased**?
  - 2 times bigger x meant the program
    - Took approx. 4 times as long to run
    - Did approx. 4 times as many ops
  - 10 times bigger x meant the program
    - Took approx. 100 times as long to run
    - Did approx. 100 times as many ops
  - Express it in an “order of” way vs. the input variable: **efficiency = Order of x^2**

ORDER of GROWTH

ORDERS OF GROWTH

- It's a notation
- Evaluates programs when **input is very big**
- Expresses the **growth of program's run time**
- Puts an **upper bound** on growth
- Do not need to be precise: **"order of" not "exact"** growth

- Focus on the **largest factors** in run time (which section of the program will take the longest to run?)

A BETTER WAY
A GENERALIZED WAY WITH APPROXIMATIONS

- Use the idea of counting operations in an algorithm, but **not worry about small variations in implementation**
  - When x is big, 3x+4 and 3x and x are pretty much the same!
  - Don't care about exact value: ops = 1+x(2+1)
  - Express it in an **"order of" way vs. the input**: ops = Order of x
- Focus on how algorithm performs when **size of problem gets arbitrarily large**
- **Relate time** needed to complete a computation **against the size of the input** to the problem
- Need to decide what to measure. What is the input?

WHICH INPUT TO USE TO MEASURE EFFICIENCY

- Want to express efficiency in terms of input, so need to **decide what is your input**

- Could be an **integer**
  -- `convert_to_km(x)`
- Could be **length of list**
  -- `list_sum(L)`
- **You decide** when multiple parameters to a function
  -- `is_in(L, e)`
  - Might be different depending on which input you consider

DIFFERENT INPUTS CHANGE HOW THE PROGRAM RUNS

- A function that searches for an element in a list

```
def is_in(L, e):
    for i in L:
        if i == e:
            return True
    return False
```

- Does the program take longer to run **as e increases**?
  - No

```
is_in([1,2,3], 0) vs.
is_in([1,2,3], 1000)
```

DIFFERENT INPUTS CHANGE HOW THE PROGRAM RUNS

- A function that searches for an element in a list

```
def is_in(L, e):
    for i in L:
        if i == e:
            return True
    return False
```

is_in([1,2,3], 0) vs.
is_in([1000,2000,3000], 0)

- Does the program take longer to run as `L` increases?
  - What if L has a fixed length and **its elements are big numbers**?
    - No
  - What if L has **different lengths**?
    - Yes!

is_in([1,2,3], 0) vs.
is_in([1,2,3,4,5,6,7,8,9,10], 0)

DIFFERENT INPUTS CHANGE HOW THE PROGRAM RUNS

- A function that searches for an element in a list

```
def is_in(L, e):
    for i in L:
        if i == e:
            return True
    return False
```

- When `e` is **first element** in the list
  → BEST CASE
- When **look through about half** of the elements in list
  → AVERAGE CASE
- When `e` is **not in list**
  → WORST CASE
  - Want to measure this behavior in a general way

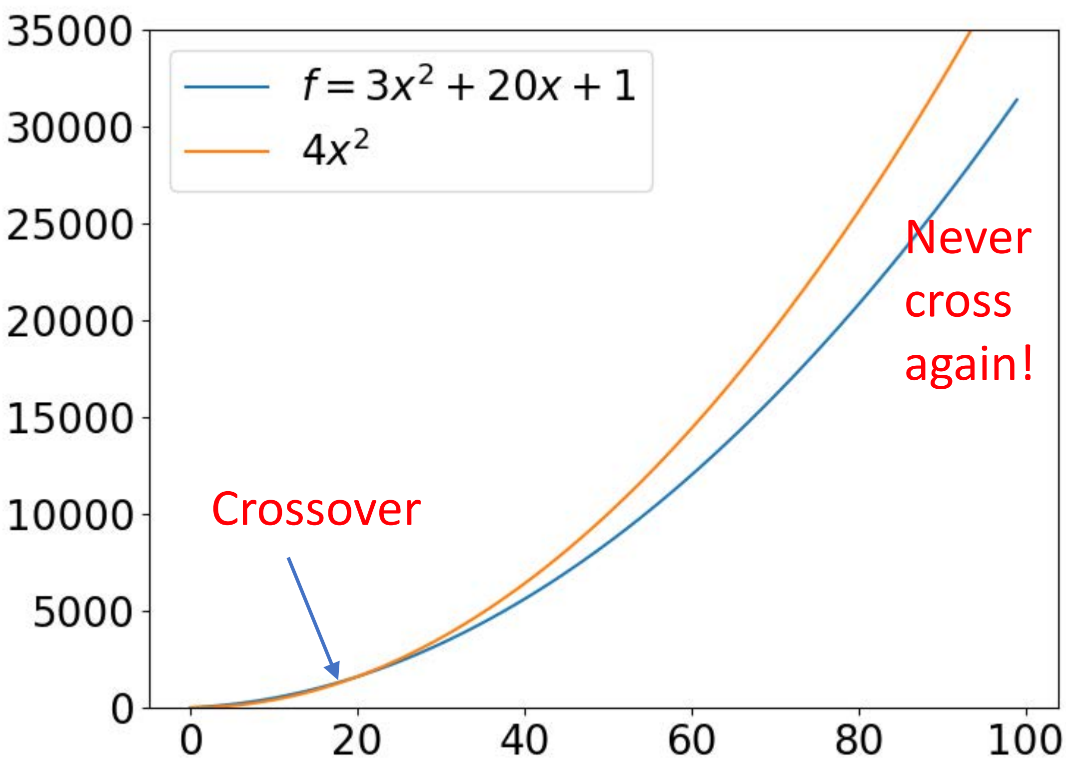
ASYMPTOTIC GROWTH

- Goal: describe how time grows as size of input grows
  - Formula relating input to number of operations
- Given an expression for the number of operations needed to compute an algorithm, want to know **asymptotic behavior as size of problem gets large**
  - Want to put a **bound** on growth
  - Do not need to be precise: **"order of" not "exact"** growth
- Will focus on term that grows most rapidly
  - Ignore additive and multiplicative constants, since want to know how rapidly time required increases as we increase size of input
- This is called ***order of growth***
  - Use mathematical notions of **"big O"** and **"big Θ"**

Big Oh and **Big Theta**

BIG O Definition

$$\boxed{3x^2 + 20x + 1 = O(x^2)}$$



$\boxed{4x^2} > 3x^2 + 20x + 1 \forall x > \boxed{20.04}$

- Suppose some code runs in $f(x) = 3x^2 + 20x + 1$ steps
  - Think of this as the formula from counting the number of ops.
- Big OH is a way to upper bound the growth of *any* function
- f(x) = O(g(x)) means that g(x) times *some* constant *eventually* always exceeds f(x)
  - *Eventually* means above some threshold value of x

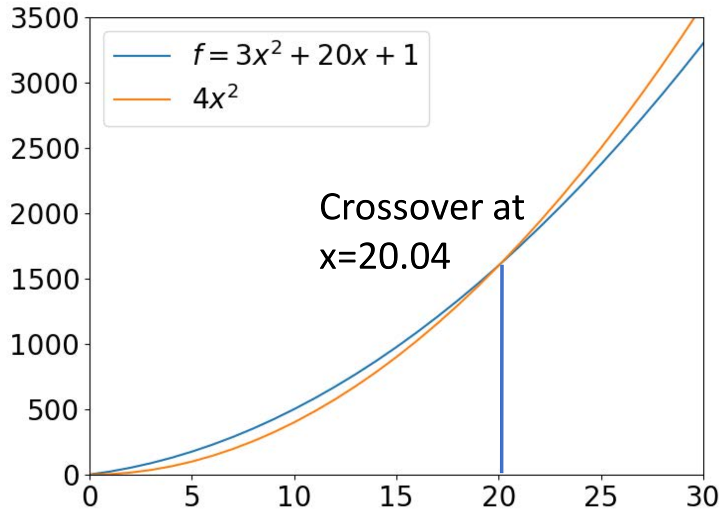
BIG O FORMALLY

- A big Oh bound is an **upper bound** on the growth of some function
- $\boxed{f(x) = O(g(x))}$ means there exist constants c_0, x_0 for which $\boxed{c_0}\boxed{g(x)} \geq \boxed{f(x)}$ for all $x > \boxed{x_0}$

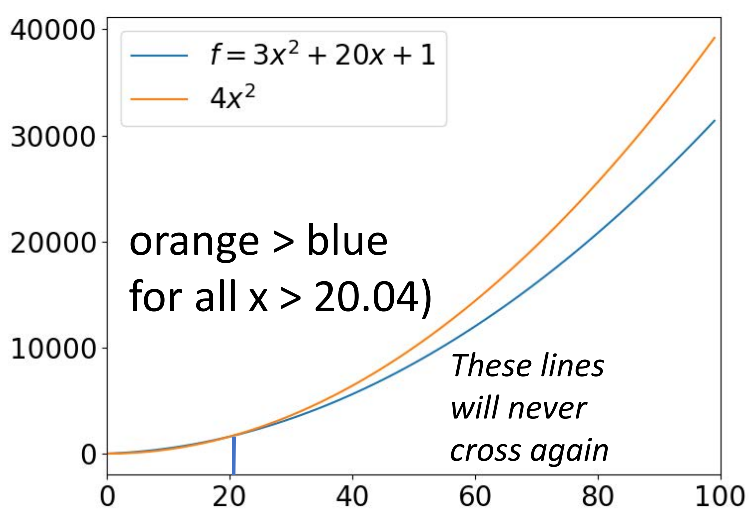
Example: $f(x) = 3x^2 + 20x + 1$

$\boxed{f(x) = O(x^2)}$,because $\boxed{4}\boxed{x^2} > \boxed{\mathbf{3x^2 + 20x + 1}} \forall x \geq \boxed{\mathbf{21}}$

$(c_0 = 4, x_0 = 20.04)$



0 <= x <= 30



0 <= x <= 100

BIG Θ Definition

$$3x^2 - 20x - 1 = \theta(x^2)$$

- A **big Θ** bound is **a lower and upper bound** on the growth of some function

Suppose $f(x) = 3x^2 - 20x - 1$

$\boldsymbol{f}(\boldsymbol{x}) = \Theta(\boldsymbol{g}(\boldsymbol{x}))$ means:

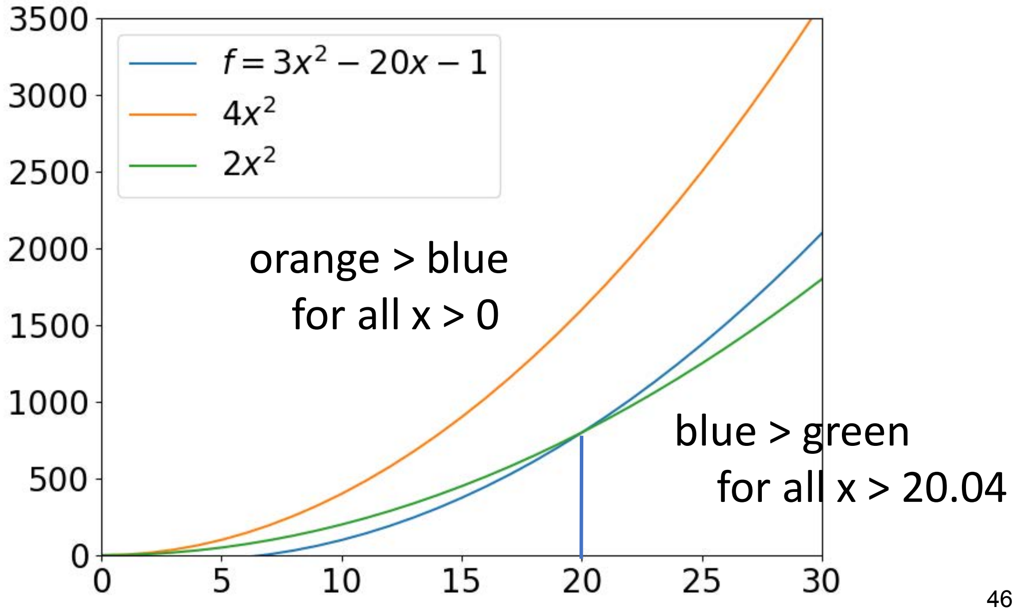
there exist constants $\boldsymbol{c_0}, x_0$ for which $\boxed{\boldsymbol{c_0} \boldsymbol{g}(\boldsymbol{x}) \geq \boldsymbol{f}(\boldsymbol{x}) \text{ for all } x > \boldsymbol{x_0}}$

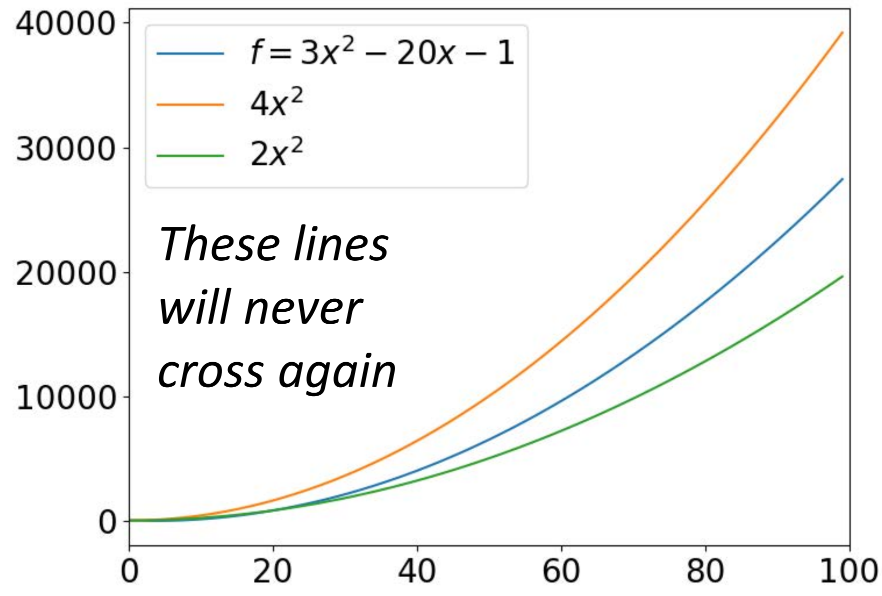
and constants $\boldsymbol{c_1}, x_1$ for which $\boldsymbol{c_1} \boldsymbol{g}(\boldsymbol{x}) \leq \boldsymbol{f}(\boldsymbol{x})$ for all $x > \boldsymbol{x_1}$

Big O definition

- Example, $\boldsymbol{f}(\boldsymbol{x}) = \Theta(\boldsymbol{x^2})$ because $\boxed{\mathbf{4x^2 > 3x^2 - 20x - 1} \ \forall x \geq \mathbf{0} \quad (\boldsymbol{c_0} = \mathbf{4}, \boldsymbol{x_0} = \mathbf{0})}$

  and $\mathbf{2x^2 < 3x^2 - 20x - 1} \ \forall x \geq \mathbf{21} \quad (\boldsymbol{c_1} = \mathbf{2}, \boldsymbol{x_1} = \mathbf{20.04})$





0 <= x <= 100

Θ vs O

- In practice, Θ bounds are preferred, because they are "tight"
  For example: $f(x) = 3x^2 - 20x - 1$

- $f(x) = O(x^2) = O(x^3) = O(2^x)$ and anything higher order because they all upper bound it

- $\boldsymbol{f(x) = \Theta(x^2)}$
  $\neq \Theta(x^3) \neq \Theta(2^x)$ and anything higher order because they upper bound but not lower bound it

SIMPLIFICATION EXAMPLES

- Drop constants and multiplicative factors
- Focus on **dominant term**

$\Theta(n^2)$: $\boxed{n^2} + 2n + 2$

$\Theta(x^2)$: $\boxed{3x^2} + 100000x + 3^{1000}$

$\Theta(a)$: $\log(a) + \boxed{a} + 4$

BIG IDEA

Express Theta in terms of the input.

Don't just use n all the time!

YOU TRY IT!

$\Theta(x)$: `1000*log(x) + x`

$\Theta(n^3)$: $n^2\log(n) + n^3$

$\Theta(y)$: `log(y) + 0.000001y`

$\Theta(2^b)$: $2^b + 1000a^2 + 100*b^2 + 0.0001a^3$

$\Theta(a^3)$

$\Theta(2^b+a^3)$

All could be ok, depends on the input we care about

USING Θ TO EVALUATE YOUR ALGORITHM

```
def fact_iter(n):
    """assumes n an int >= 0"""
    answer = 1
    while n > 1:
        answer *= n
        n -= 1
    return answer
```

5 steps inside loop
1. compare,
2. multiply,
3. assign,
4. subtract,
5. assign

- Number of steps: $5n + 2$
- Worst case asymptotic complexity: $\Theta(n)$
  - Ignore additive constants
    - 2 doesn't matter when n is big
  - Ignore multiplicative constants
    - 5 doesn't matter if just want to know how increasing n changes time needed

COMBINING COMPLEXITY CLASSES
LOOPS IN SERIES

- Analyze statements inside functions to get order of growth
- Apply some rules, focus on dominant term

- **Law of Addition** for Θ():
  - Used with **sequential** statements
  - $\Theta(f(n)) + \Theta(g(n)) = \Theta(f(n) + g(n))$
- For example,

```
for i in range(n):        Θ(n)
    print('a')
for j in range(n*n):      Θ(n²)
    print('b')
```

is $\Theta(n) + \Theta(n * n) = \Theta(n + n^2) = \Theta(n^2)$ because of dominant n^2 term

COMBINING COMPLEXITY CLASSES NESTED LOOPS

- Analyze statements inside functions to get order of growth
- Apply some rules, focus on dominant term

- **Law of Multiplication for Θ():**
  - Used with **nested** statements/loops
  - $\Theta(f(n)) * \Theta(g(n)) = \Theta(f(n) * g(n))$
- For example,

```
for i in range(n):
    for j in range(n//2):
        print('a')
```

Θ(n)

Θ(n) for each outer loop iteration

- $\Theta(n) \times \Theta(n) = \Theta(n \times n) = \Theta(n^2)$
  - Outer loop runs n times and the inner loop runs n times for every outer loop iteration.

ANALYZE COMPLEXITY

- What is the Theta complexity of this program?

Always note the input parameter!

```
def f(x):
    answer = 1
    for i in range(x):
        for j in range(i,x):
            answer += 2
    return answer
```

Outer loop is Θ(x)
Inner loop is Θ(x)
Everything else is Θ(1)

- $\boxed{\Theta(1)} + \boxed{\Theta(x) * \Theta(x) * \Theta(1)} + \boxed{\Theta(1)}$
- Overall complexity is **$\Theta(x^2)$** by rules of addition and multiplication

YOU TRY IT!

- What is the Theta complexity of this program? Careful to describe in terms of input
(hint: what matters with a list, size of elems of length?)

```
def f(L):
    Lnew = []
    for i in L:
        Lnew.append(i**2)
    return Lnew
```

ANSWER:

Loop: Θ(len(L))

f is Θ(len(L))

YOU TRY IT!

- What is the Theta complexity of this program?

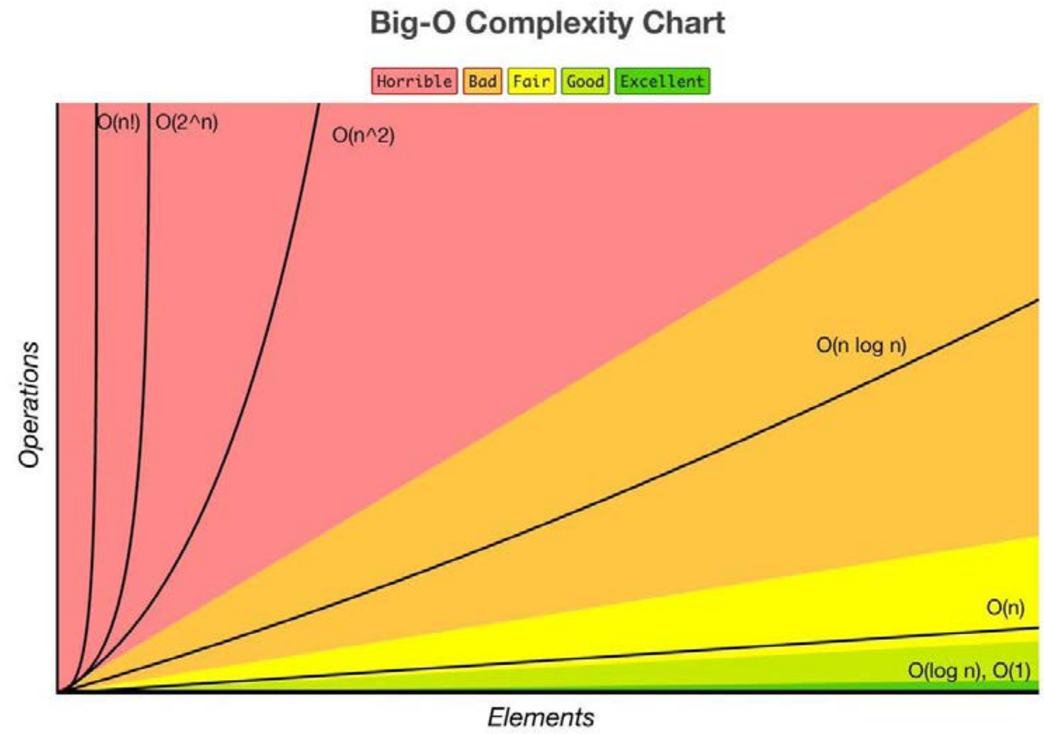
```
def f(L, L1, L2):
    """ L, L1, L2 are the same length """
    inL1 = False
    for i in range(len(L)):
        if L[i] == L1[i]:
            inL1 = True
    inL2 = False
    for i in range(len(L)):
        if L[i] == L2[i]:
            inL2 = True
   return inL1 and inL2
```

ANSWER:

Loop: Θ(len(L)) + Θ(len(L))

f is Θ(len(L)) or Θ(len(L1)) or Θ(len(L2))

COMPLEXITY CLASSES



We want to design algorithms that are as close to top of this hierarchy as possible

- *Θ(1)* denotes **constant** running time
- *Θ(log n)* denotes **logarithmic** running time
- *Θ(n)* denotes **linear** running time
- *Θ(n log n)* denotes **log-linear** running time
- $\Theta(n^c)$ denotes **polynomial** running time
  (c is a constant)
- $\Theta(c^n)$ denotes **exponential** running time
  (c is a constant raised to a power based on input size)

COMPLEXITY GROWTH

CLASS	N = 10	N = 100	N = 1000	N = 1000000
Constant	1	1	1	1
Logarithmic	1	2	3	6
Linear	10	100	1000	1000000
Log-linear	10	200	3000	6000000
Polynomial	100	10000	1000000	1000000000000
Exponential	1024	12676506 00228229 40149670 3205376	1071508607186267320948425 0490600018105614048117055 3360744375038837035105112 4936122493198378815695858 1275946729175531468251871 4528569231404359845775746 9857480393456777482423098 5421074605062371141877954 1821530464749835819412673 9876755916554394607706291 4571196477686542167660429 8316526243868372056680693 76	Good Luck!!

SUMMARY

- Timing is machine/implementation/algorithm dependent
- Counting ops is implementation/algorithm dependent
- Order of growth is algorithm dependent

- Compare **efficiency of algorithms**
  - Notation that describes growth
  - **Lower order of growth** is better
  - Independent of machine or specific implementation
- Using Theta
  - Describe asymptotic order of growth
  - **Asymptotic notation**
  - **Upper bound** and a **lower bound**

COMPLEXITY CLASSES EXAMPLES

(download slides and .py files to follow along)

6.100L Lecture 23

Ana Bell

THETA

- **Theta Θ** is how we denote the **asymptotic complexity**

- We look at the **input term that dominates** the function
  - Drop other pieces that don't have the fastest growth
  - Drop additive constants
  - Drop multiplicative constants

- End up with only a **few classes of algorithms**
- We will look at code that lands in each of these classes today

WHERE DOES THE FUNCTION COME FROM?

- Given code, start with the input parameters. What are they?
- Come up with the equation relating input to number of ops.
  - `f = 1 + len(L1)*5 + 1 + len(L2)*5 + 2 = 5*len(L1) + 5*len(L2) + 3`
  - If lengths are the same, `f = 10*len(L) + 3`
- Θ(f) = Θ (10*len(L) + 3) = Θ(len(L))

Only care about code that repeats wrt these variables

```
def f(L, L1, L2):
    inL1 = False
    for i in range(len(L1)):
        if L[i] == L1[i]:
            inL1 = True
    inL2 = False
    for i in range(len(L2)):
        if L[i] == L2[i]:
            inL2 = True
    return inL1 and inL2
```

Loop repeats as a function of input

Loop repeats as a function of input

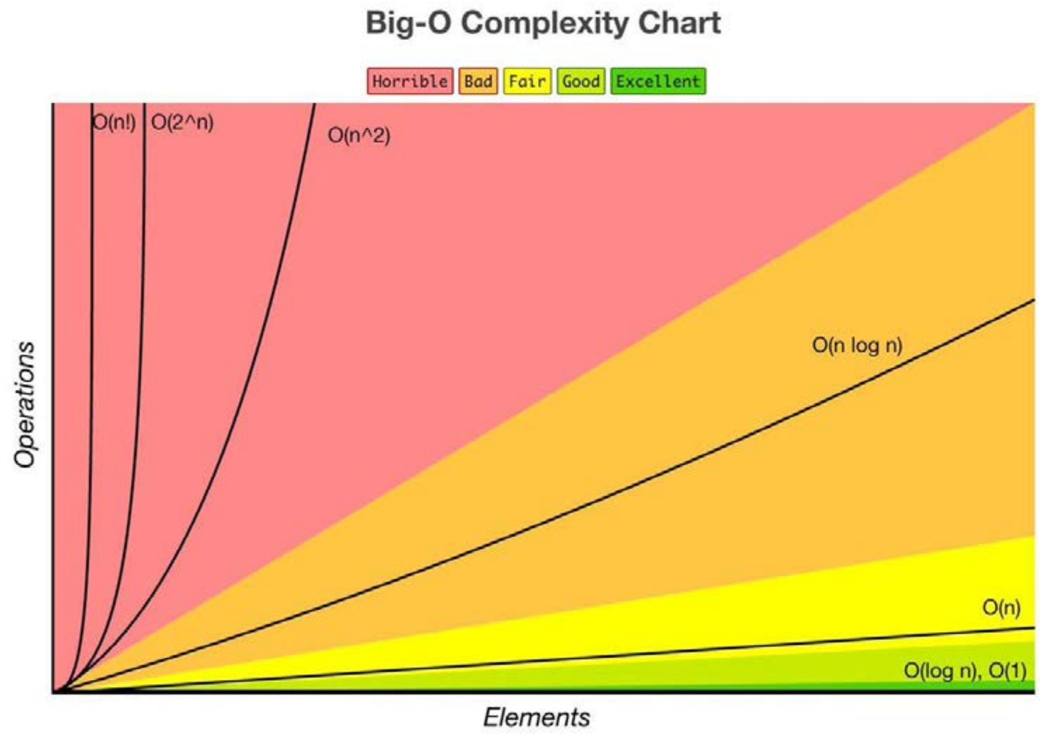
WHERE DOES THE FUNCTION COME FROM?

- A quicker way: no need to come up with the exact formula. Look for loops and anything that repeats wrt the input parameters. Everything else is constant.

Only care about code that repeats wrt these variables

```
def f(L, L1, L2):
    inL1 = False
    for i in range(len(L1)):
        if L[i] == L1[i]:
            inL1 = True
    inL2 = False
    for i in range(len(L2)):
        if L[i] == L2[i]:
            inL2 = True
  return inL1 and inL2
```

COMPLEXITY CLASSES
n is the input



We want to design algorithms that are as close to top of this hierarchy as possible

- *Θ(1)* denotes **constant** running time
- *Θ(log n)* denotes **logarithmic** running time
- *Θ(n)* denotes **linear** running time
- *Θ(n log n)* denotes **log-linear** running time
- $\Theta(n^c)$ denotes **polynomial** running time
  (c is a constant)
- $\Theta(c^n)$ denotes **exponential** running time
  (c is a constant raised to a power based on input size)

CONSTANT COMPLEXITY

CONSTANT COMPLEXITY

- Complexity **independent of inputs**
- Very few interesting algorithms in this class, but can often have pieces that fit this class
- **Can have loops or recursive calls**, but number of iterations or calls independent of size of input
- Some built-in operations to a language are constant
  - Python indexing into a list `L[i]`
  - Python list append `L.append()`
  - Python dictionary lookup `d[key]`

CONSTANT COMPLEXITY: EXAMPLE 1

```
def add(x, y):
    return x+y
```

- Complexity in terms of either x or y: **$\Theta(1)$**

CONSTANT COMPLEXITY: EXAMPLE 2

```
def convert_to_km(m):
    return m*1.609
```

- Complexity in terms of m: **Θ(1)**

CONSTANT COMPLEXITY: EXAMPLE 3

```
def loop(x):
    y = 100
    total = 0
    for i in range(y):
        total += x
    return total
```

- Complexity in terms of x (the input parameter): **Θ(1)**

LINEAR COMPLEXITY

LINEAR COMPLEXITY

- Simple **iterative loop** algorithms
  - Loops must be a **function of input**
- Linear search a list to see if an element is present
- Recursive functions with **one recursive call and constant overhead** for call
- Some built-in operations are linear
  - `e in L`
  - Subset of list: e.g. `L[:len(L)//2]`
  - `L1 == L2`
  - `del(L[5])`

COMPLEXITY EXAMPLE 0 (with a twist)

- Multiply x by y

```
def mul(x, y):
    tot = 0
    for i in range(y):
        tot += x
    return tot
```

Choice of input on which to measure complexity matters

- Complexity in terms of y: **Θ(y)**
- Complexity in terms of x: **Θ(1)**

BIG IDEA

Be careful about what the inputs are.

LINEAR COMPLEXITY: EXAMPLE 1

- Add characters of a string, assumed to be composed of decimal digits

```
def add_digits(s):
    val = 0
    for c in s:
        val += int(c)
    return val
```

Loop goes through len(s) times: **Θ(len(s))**

Everything else is constant. **Θ(1)**

- **Θ(len(s))**
- **Θ(n) where n is len(s)**

LINEAR COMPLEXITY: EXAMPLE 2

- Loop to find the factorial of a number >=2

```python
def fact_iter(n):
    prod = 1
    for i in range(2, n+1):
        prod *= i
    return prod
```

Loop goes through n-1 times:
Θ(n)

Everything else is constant.
Θ(1)

- Number of times around loop is n-1
- Number of operations inside loop is a constant
  - Independent of n
- Overall just **Θ(n)**

FUNNY THING ABOUT FACTORIAL AND PYTHON

```
iter fact(40) took 3.10e-06 sec (322,580.65/sec)
iter fact(80) took 6.00e-06 sec (166,666.67/sec)
iter fact(160) took 1.34e-05 sec (74,626.87/sec)
iter fact(320) took 3.39e-05 sec (29,498.53/sec)
iter fact(640) took 1.18e-04 sec (8,488.96/sec)
iter fact(1280) took 4.31e-04 sec (2,322.88/sec)
iter fact(2560) took 1.33e-03 sec (752.73/sec)
iter fact(5120) took 4.94e-03 sec (202.24/sec)
iter fact(10240) took 1.90e-02 sec (52.50/sec)
iter fact(20480) took 7.66e-02 sec (13.06/sec)
iter fact(40960) took 3.35e-01 sec (2.99/sec)
iter fact(81920) took 1.60e+00 sec (0.62/sec)
```

- Eventually grows faster than linear
- Because Python increases the size of integers, which yields more costly operations
- For this class: ignore such effects

LINEAR COMPLEXITY: EXAMPLE 3

```
def fact_recur(n):
    """ assume n >= 0 """
    if n <= 1:
        return 1
    else:
        return n*fact_recur(n - 1)
```

Think about the function call stack: **Θ(n)**

Everything else is constant. **Θ(1)**

- Computes factorial recursively
- If you time it, notice that it runs a bit slower than iterative version due to function calls
- **Θ(n)** because the number of function calls is linear in n
- **Iterative and recursive factorial** implementations are the **same order of growth**

LINEAR COMPLEXITY: EXAMPLE 4

```
def compound(invest, interest, n_months):
    total=0
    for i in range(n_months):
        total = total * interest + invest
    return total
```

Θ(n_months)

Θ(1)

- Θ(1)*Θ(n_months) = **Θ(n_months)**
  Θ(n) where n=n_months
  - If I was being thorough, then need to account for assignment and return statements:
  - Θ(1) + 4*Θ(n) + Θ(1) = Θ(1 + 4*n + 1) = **Θ(n) where n=n_months**

COMPLEXITY OF ITERATIVE FIBONACCI

```
def fib_iter(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        fib_i = 0
        fib_ii = 1
        for i in range(n-1):
            tmp = fib_i
            fib_i = fib_ii
            fib_ii = tmp + fib_ii
        return fib_ii
```

constant
$\Theta(1)$

constant
$\Theta(1)$

linear
$\Theta(n)$

constant
$\Theta(1)$

$\Theta(1) + \Theta(1) + \Theta(n) * \Theta(1) + \Theta(1)$

$\rightarrow$ **$\Theta(n)$**

POLYNOMIAL COMPLEXITY

POLYNOMIAL COMPLEXITY (OFTEN QUADRATIC)

- Most **common polynomial algorithms are quadratic**, i.e., complexity grows with square of size of input
- Commonly occurs when we have **nested loops** or recursive function calls

QUADRATIC COMPLEXITY: EXAMPLE 1

```
def g(n):
    """ assume n >= 0 """
    x = 0
    for i in range(n):
        for j in range(n):
            x += 1
    return x
```

Outer loop goes through n times: **Θ(n)**

Inner loop goes through n times: **Θ(n)**

Everything else is constant. **Θ(1)**

- Computes n^2 very inefficiently
- Look at the loops. Are they **in terms of the input**?
  - Nested loops
  - Look at the ranges
  - Each iterating n times
- $\Theta(n) * \Theta(n) * \Theta(1) = \mathbf{\Theta(n^2)}$

QUADRATIC COMPLEXITY: EXAMPLE 2

- Decide if L1 is a subset of L2: are all elements of L1 in L2?

Yes:

```
L1 = [3, 5, 2]
L2 = [2, 3, 5, 9]
```

No:

```
L1 = [3, 5, 2]
L2 = [2, 5, 9]
```

```
def is_subset(L1, L2):
    for e1 in L1:
        matched = False
        for e2 in L2:
            if e1 == e2:
                matched = True
                break
        if not matched:
            return False
    return True
```

QUADRATIC COMPLEXITY: EXAMPLE 2

```
def is_subset(L1, L2):
    for e1 in L1:
        matched = False
        for e2 in L2:
            if e1 == e2:
                matched = True
                break
        if not matched:
            return False
    return True
```

Outer loop executed len(L1) times

Each iteration will execute inner loop up to to len(L2) times

Θ(len(L1)*len(L2))

If L1 and L2 same length and none of elements of L1 in L2

Θ(len(L1)^2)

QUADRATIC COMPLEXITY: EXAMPLE 3

- Find intersection of two lists, return a list with each element appearing only once
  Example:

```
L1 = [3, 5, 2]
L2 = [2, 3, 5, 9]
returns [2,3,5]
```

```
L1 = [7, 7, 7]
L2 = [7, 7, 7]
returns [7]
```

```
def intersect(L1, L2):
    tmp = []
    for e1 in L1:
        for e2 in L2:
            if e1 == e2:
                tmp.append(e1)
    unique = []
    for e in tmp:
        if not(e in unique):
            unique.append(e)
    return unique
```

Build the list with common elements in L1 and L2. May have dups

Keep only unique values

QUADRATIC COMPLEXITY: EXAMPLE 3

```
def intersect(L1, L2):
    tmp = []
    for e1 in L1:
        for e2 in L2:
            if e1 == e2:
                tmp.append(e1)
    unique = []
    for e in tmp:
        if not(e in unique):
            unique.append(e)
    return unique
```

First nested loop takes **Θ(len(L1)*len(L2))** steps.

Second loop takes at most **Θ(len(L1)*len(L2))** steps. Typically not this bad.

- E.g: [7,7,7] and [7,7,7] makes tmp=[7,7,7,7,7,7,7,7,7]

Overall **Θ(len(L1)*len(L2))**

DIAMETER COMPLEXITY

Outer loop does len(L) passes:
Θ(len(L))

Inner loop does len(L) / 2 passes
(on average): **Θ(len(L))**

Everything else is constant **Θ(1)**

```
def diameter(L):
    farthest_dist = 0
    for i in range(len(L)):
        for j in range(i+1, len(L)):
            p1 = L[i]
            p2 = L[j]
            dist = math.sqrt( (p1[0]-p2[0])**2 + (p1[1]-p2[1])**2 )
            if dist > farthest_dist:
                farthest_dist = dist
    return farthest_dist
```

len(L) * len(L)/2 iterations = $\text{len}(L)^2 / 2$

$\Theta(\text{len}(L)^2)$

YOU TRY IT!

```
def all_digits(nums):
    """ nums is a list of numbers """
    digits = [0,1,2,3,4,5,6,7,8,9]
    for i in nums:
        isin = False
        for j in digits:
            if i == j:
                isin = True
                break
        if not isin:
            return False
    return True
```

ANSWER:

What's the input?

Outer for loop is Θ(nums).

Inner for loop is Θ(1).

Overall: Θ(len(nums))

YOU TRY IT!

- Asymptotic complexity of f? And if L1,L2,L3 are same length?

```
def f(L1, L2, L3):
    for e1 in L1:
        for e2 in L2:
            if e1 in L3 and e2 in L3 :
                return True
    return False
```

ANSWER:

Θ(len(L1))* Θ(len(L2))* Θ(len(L3)+len(L3))

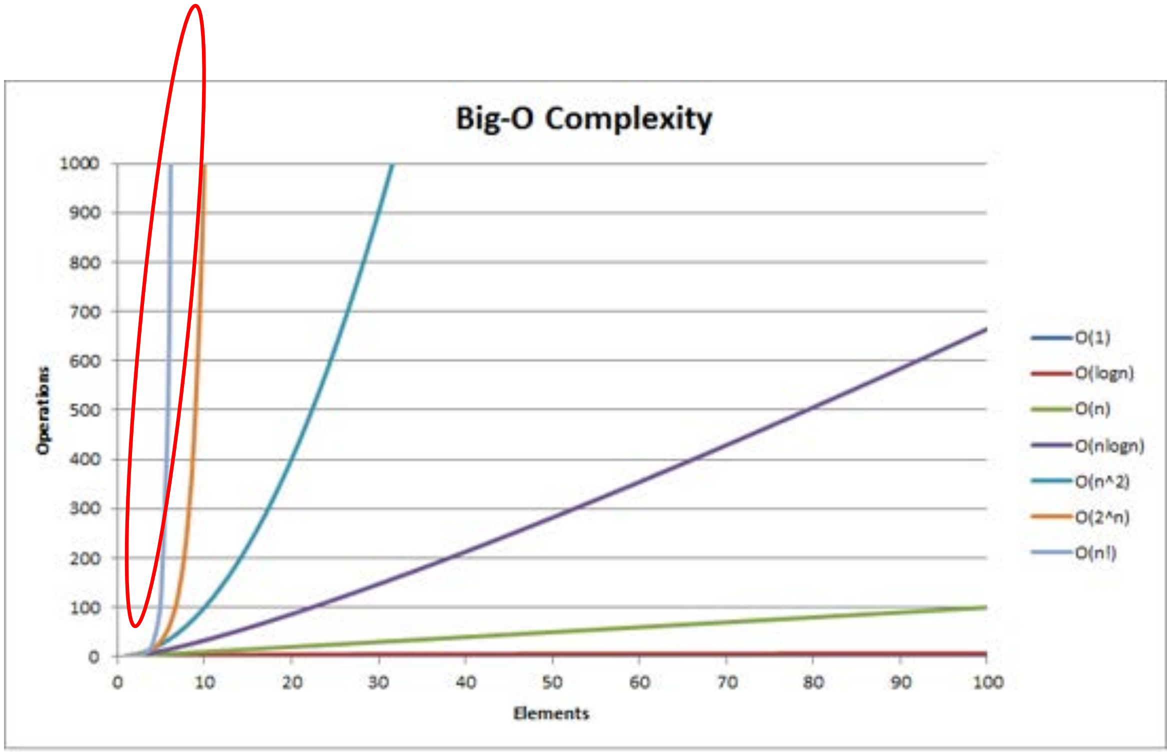
Overall: Θ(len(L1)*len(L2)*len(L3))

Overall if lists equal length: Θ(len(L1)**3)

EXPONENTIAL COMPLEXITY

EXPONENTIAL COMPLEXITY

- Recursive functions where have more than one recursive call for each size of problem
  - Fibonacci
- Many important problems are inherently exponential
  - Unfortunate, as cost can be high
  - Will lead us to consider approximate solutions more quickly



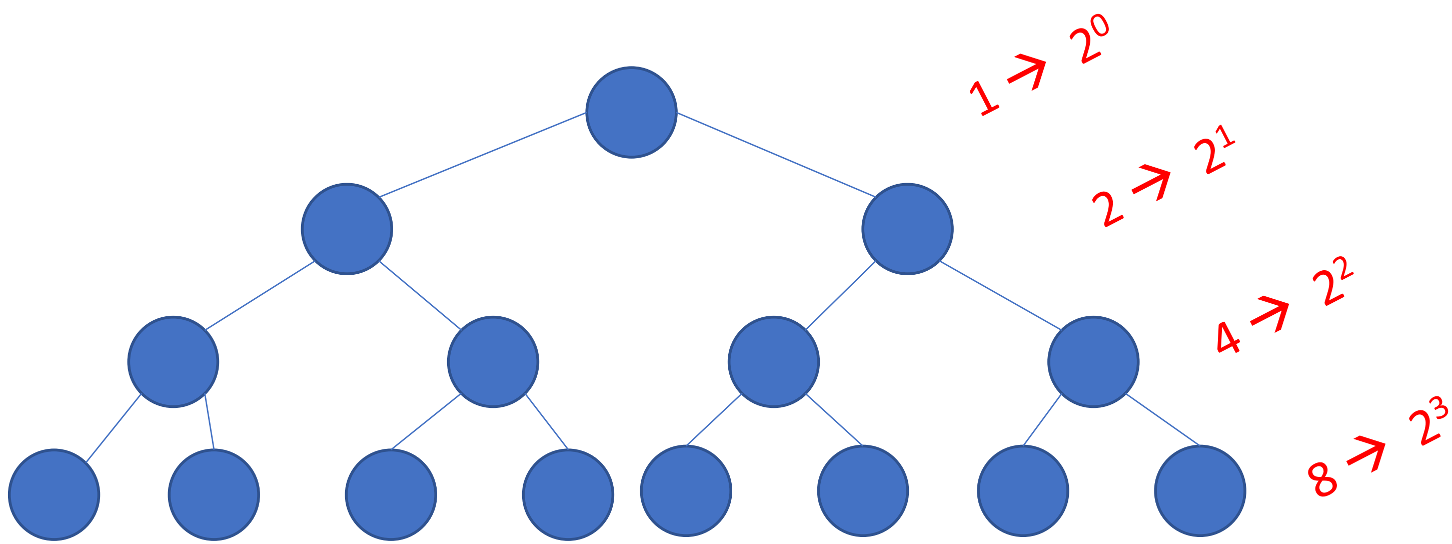
2^{30} ~= 1 million
2^{100} > # cycles than all the computers in the world working for all of recorded history could complete

COMPLEXITY OF RECURSIVE FIBONACCI

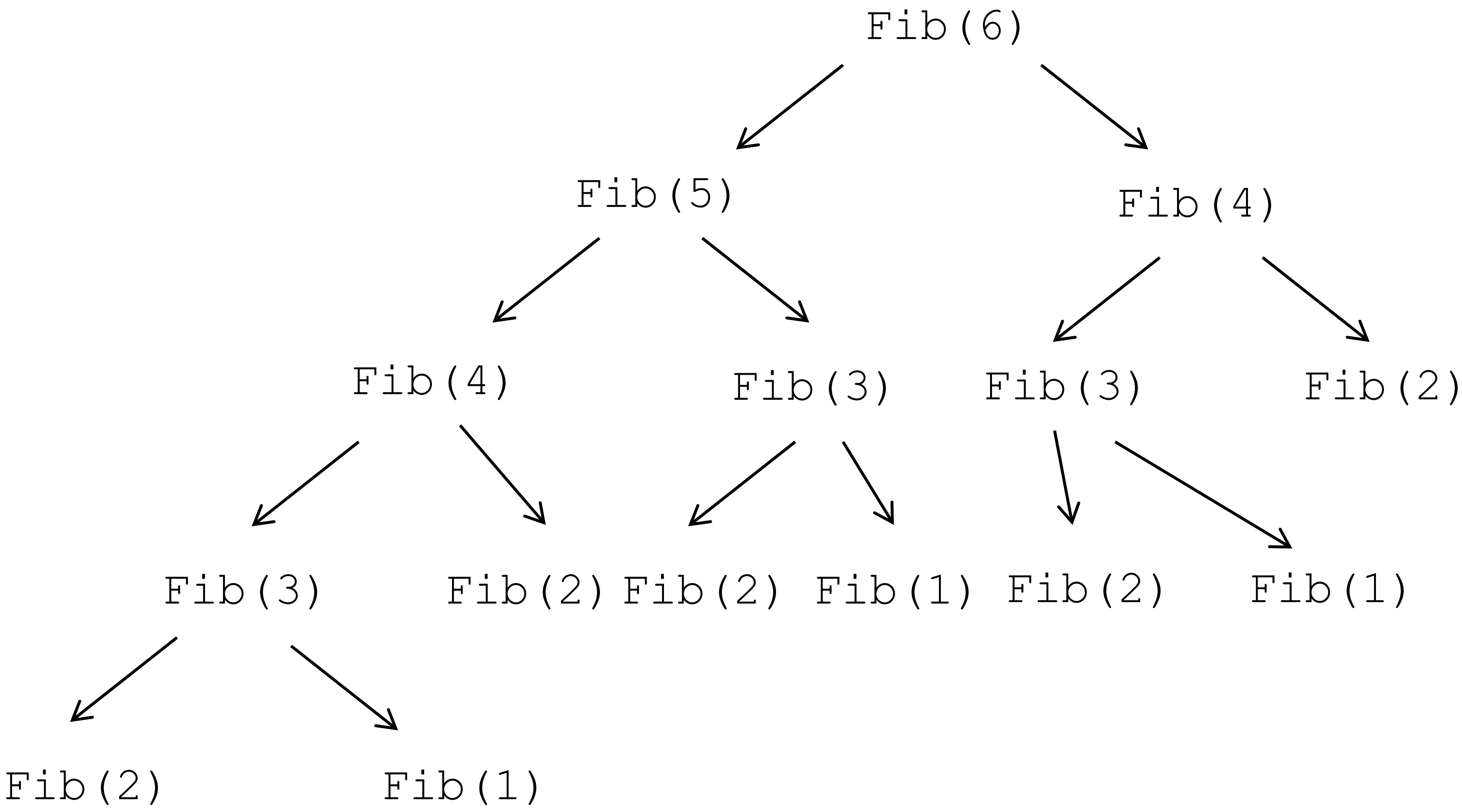
```
def fib_recur(n):
    """ assumes n an int >= 0 """
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib_recur(n-1) + fib_recur(n-2)
```

- Worst case:

$\Theta(2^n)$



COMPLEXITY OF RECURSIVE FIBONACCI



- Can do a bit better than 2^n since tree thins out to the right
- But complexity is still order exponential

EXPONENTIAL COMPLEXITY: GENERATE SUBSETS

- Input is `[1, 2, 3]`
- Output is all combinations of elements of all lengths
  `[[],[1],[2],[3],[1,2],[1,3],[2,3],[1,2,3]]`

```
def gen_subsets(L):
    if len(L) == 0:
        return [[]]
    extra = L[-1:]
    smaller = gen_subsets(L[:-1])
    new = []
    for small in smaller:
        new.append(small+extra)
    return smaller+new
```

Base case: reach list of empty list

Create a list of just last element

All subsets without last element

For all smaller solutions, add one with last element

Combine those with last element and those without

VISUALIZING the ALGORITHM

Extra is [3]

[1,2,$\boxed{3}$]

Extra is [2]

[1,$\boxed{2}$]

Extra is [1]

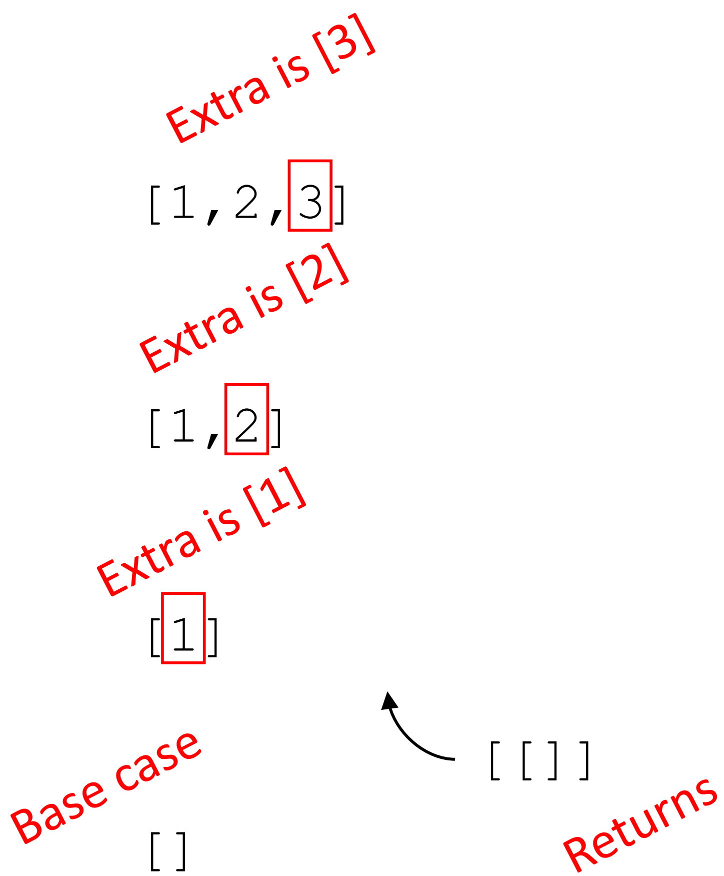
[$\boxed{1}$]

Base case

[]

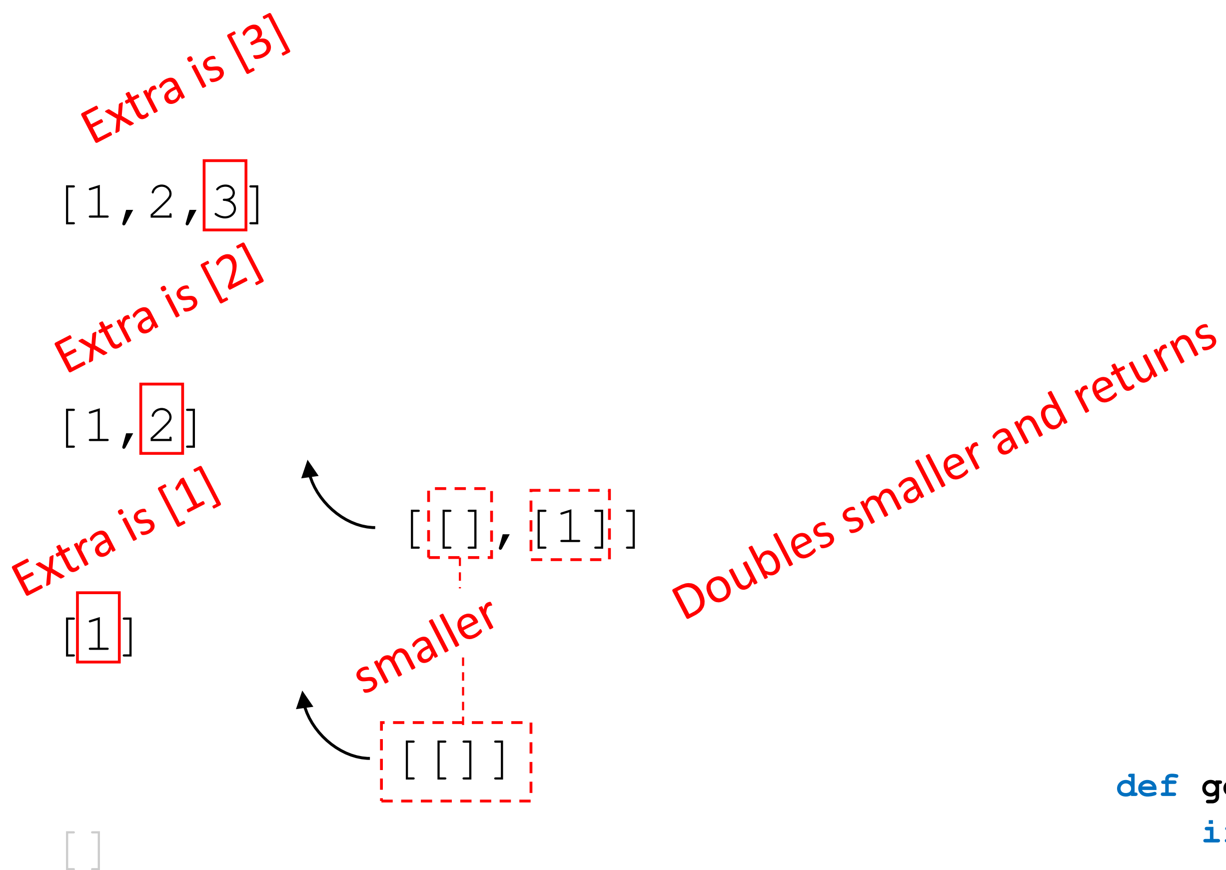
```
def gen_subsets(L):
    if len(L) == 0:
        return [[]]
    extra = L[-1:]
    smaller = gen_subsets(L[:-1])
    new = []
    for small in smaller:
        new.append(small+extra)
    return smaller+new
```

VISUALIZING the ALGORITHM




```python
def gen_subsets(L):
    if len(L) == 0:
        return [[]]
    extra = L[-1:]
    smaller = gen_subsets(L[:-1])
    new = []
    for small in smaller:
        new.append(small+extra)
    return smaller+new
```

VISUALIZING the ALGORITHM




```python
def gen_subsets(L):
    if len(L) == 0:
        return [[]]
    extra = L[-1:]
    smaller = gen_subsets(L[:-1])
    new = []
    for small in smaller:
        new.append(small+extra)
    return smaller+new
```

VISUALIZING the ALGORITHM

Extra is [3]

[1,2,3]

Extra is [2]

[[],[1],[2],[1,2]]

[1,2]

smaller

[[],[1]]

Doubles smaller and returns

[1]

[[]]

[]

```python
def gen_subsets(L):
    if len(L) == 0:
        return [[]]
    extra = L[-1:]
    smaller = gen_subsets(L[:-1])
    new = []
    for small in smaller:
        new.append(small+extra)
    return smaller+new
```

VISUALIZING the ALGORITHM

Extra is [3]

[[],[1],[2],[1,2],[3],[1,3],[2,3],[1,2,3]]

[1,2,3]

smaller

[[],[1],[2],[1,2]]

Doubles smaller and returns

[1,2]

[[],[1]]

[1]

[[]]

[]

```python
def gen_subsets(L):
    if len(L) == 0:
        return [[]]
    extra = L[-1:]
    smaller = gen_subsets(L[:-1])
    new = []
    for small in smaller:
        new.append(small+extra)
    return smaller+new
```

VISUALIZING the ALGORITHM

[[],[1],[2],[1,2],[3],[1,3],[2,3],[1,2,3]]

[1,2,3]

[1,2]

[[],[1]]

[1]

[[]]

[]

```
def gen_subsets(L):
    if len(L) == 0:
        return [[]]
    extra = L[-1:]
    smaller = gen_subsets(L[:-1])
    new = []
    for small in smaller:
        new.append(small+extra)
    return smaller+new
```

EXPONENTIAL COMPLEXITY GENERATE SUBSETS

```
def gen_subsets(L):
    if len(L) == 0:
        return [[]]
    extra = L[-1:]
    smaller = gen_subsets(L[:-1])
    new = []
    for small in smaller:
        new.append(small+extra)
    return smaller+new
```

- Assuming append is constant time
- Time to make sublists includes time to solve **smaller problem**, and time needed to **make a copy** of all elements in smaller problem

EXPONENTIAL COMPLEXITY GENERATE SUBSETS

```
def gen_subsets(L):
    if len(L) == 0:
        return [[]]
    extra = L[-1:]
    smaller = gen_subsets(L[:-1])
    new = []
    for small in smaller:
        new.append(small+extra)
    return smaller+new
```

- Think about **size of smaller**
  - For a set of size k there are 2^k cases, doubling the size every call
  - So to solve need $2^{n-1} + 2^{n-2} + \ldots + 2^0$ steps = $\Theta(2^n)$
- Time to **make a copy of smaller**
  - Concatenation isn't constant
  - $\Theta(n)$
- Overall complexity is **$\Theta(n*2^n)$ where n=len(L)**

LOGARITHMIC COMPLEXITY

TRICKY COMPLEXITY

```
def digit_add(n):
    """ assume n an int >= 0 """
    answer = 0
    s = str(n)
    for c in s[::-1]:
        answer += int(c)
    return answer
```

Linear **Θ(len(s))**
Loops through the length of n as a str

But what in terms of input n?

- Adds digits of a number together
  - n = 83, but the loop only iterates 2 times. Relationship?
  - n = 4271, but the loop only iterates 4 times! Relationship??

4	2	7	1

First time through loop, extract the least significant digit

1

TRICKY COMPLEXITY

```
def digit_add(n):
    """ assume n an int >= 0 """
    answer = 0
    s = str(n)
    for c in s[::-1]:
        answer += int(c)
    return answer
```

Linear Θ(len(s))

But what in terms of input n?

- Adds digits of a number together
  - n = 83, but the loop only iterates 2 times. Relationship?
  - n = 4271, but the loop only iterates 4 times! Relationship??

4 2 7

Second time through loop, extract the next least significant digit

7 + 1

TRICKY COMPLEXITY

```
def digit_add(n):
    """ assume n an int >= 0 """
    answer = 0
    s = str(n)
    for c in s[::-1]:
        answer += int(c)
    return answer
```

Linear Θ(len(s))

But what in terms of input n?

- Adds digits of a number together
  - n = 83, but the loop only iterates 2 times. Relationship?
  - n = 4271, but the loop only iterates 4 times! Relationship??

4 2

Third time through loop, extract the next least significant digit

2 + 7 + 1

TRICKY COMPLEXITY

```python
def digit_add(n):
    """ assume n an int >= 0 """
    answer = 0
    s = str(n)
    for c in s[::-1]:
        answer += int(c)
    return answer
```

Linear Θ(len(s))

But what in terms of input n?

- Adds digits of a number together
  - n = 83, but the loop only iterates 2 times. Relationship?
  - n = 4271, but the loop only iterates 4 times! Relationship??

4

Last time through loop, extract the next least significant digit

4 + 2 + 7 + 1

TRICKY COMPLEXITY

```
def digit_add(n):
    """ assume n an int >= 0 """
    answer = 0
    s = str(n)
    for c in s[::-1]:
        answer += int(c)
    return answer
```

Linear **Θ(len(s))**

But what in terms of input n?

- Adds digits of a number together
- Tricky part: iterate over **length of string**, not magnitude of n
  - Think of it like dividing n by 10 each iteration
  - $n/10^{len(s)} = 1$ (i.e. divide by 10 until there is 1 element left to add)
  - len(s) = log(n)
- **Θ(log n)** – base doesn't matter

LOGARITHMIC COMPLEXITY

- Complexity grows as log of size of one of its inputs
- Example algorithm: **binary search** of a list
- Example we'll see in a few slides: one **bisection search** implementation

LIST AND DICTIONARIES

- Must be **careful** when using built-in functions!

Lists – n is len(L)

- index $\Theta(1)$
- store $\Theta(1)$
- length $\Theta(1)$
- append $\Theta(1)$
- == $\Theta(n)$
- remove $\Theta(n)$
- copy $\Theta(n)$
- reverse $\Theta(n)$
- iteration $\Theta(n)$
- in list $\Theta(n)$

Dictionaries – n is len(d)

- index $\Theta(1)$
- store $\Theta(1)$
- length $\Theta(1)$
- delete $\Theta(1)$
- .keys $\Theta(n)$
- .values $\Theta(n)$
- iteration $\Theta(n)$

SEARCHING ALGORITHMS

SEARCHING ALGORITHMS

- Linear search
  - **Brute force** search
  - List does not have to be sorted
- Bisection search
  - List **MUST be sorted** to give correct answer
  - Will see two different implementations of the algorithm

LINEAR SEARCH ON UNSORTED LIST

```
def linear_search(L, e):
    found = False
    for i in range(len(L)):
        if e == L[i]:
            found = True
    return found
```

The loop goes through len(L): **Θ(len(L))**

Everything else is constant. **Θ(1)**

- Must look through all elements to decide it's not there
- **Θ(len(L))** for the loop * **Θ(1)** to test if e == L[i]
- Overall complexity is **Θ(n) where n is len(L)**
- **Θ(len(L))**

LINEAR SEARCH ON UNSORTED LIST

```
def linear_search(L, e):
    for i in range(len(L)):
        if e == L[i]:
            return True
    return False
```

Speed up a little by returning True here, but speed up doesn't impact worst case

- Must look through all elements to decide it's not there
- **Θ(len(L))** for the loop * **Θ(1)** to test if e == L[i]
- Overall complexity is **Θ(n) where n is len(L)**
- **Θ(len(L))**

LINEAR SEARCH ON SORTED LIST

```
def search(L, e):
    for i in L:
        if i == e:
            return True
        if i > e:
            return False
    return False
```

The loop goes through len(L):
Θ(len(L))

Everything else is constant.
Θ(1)

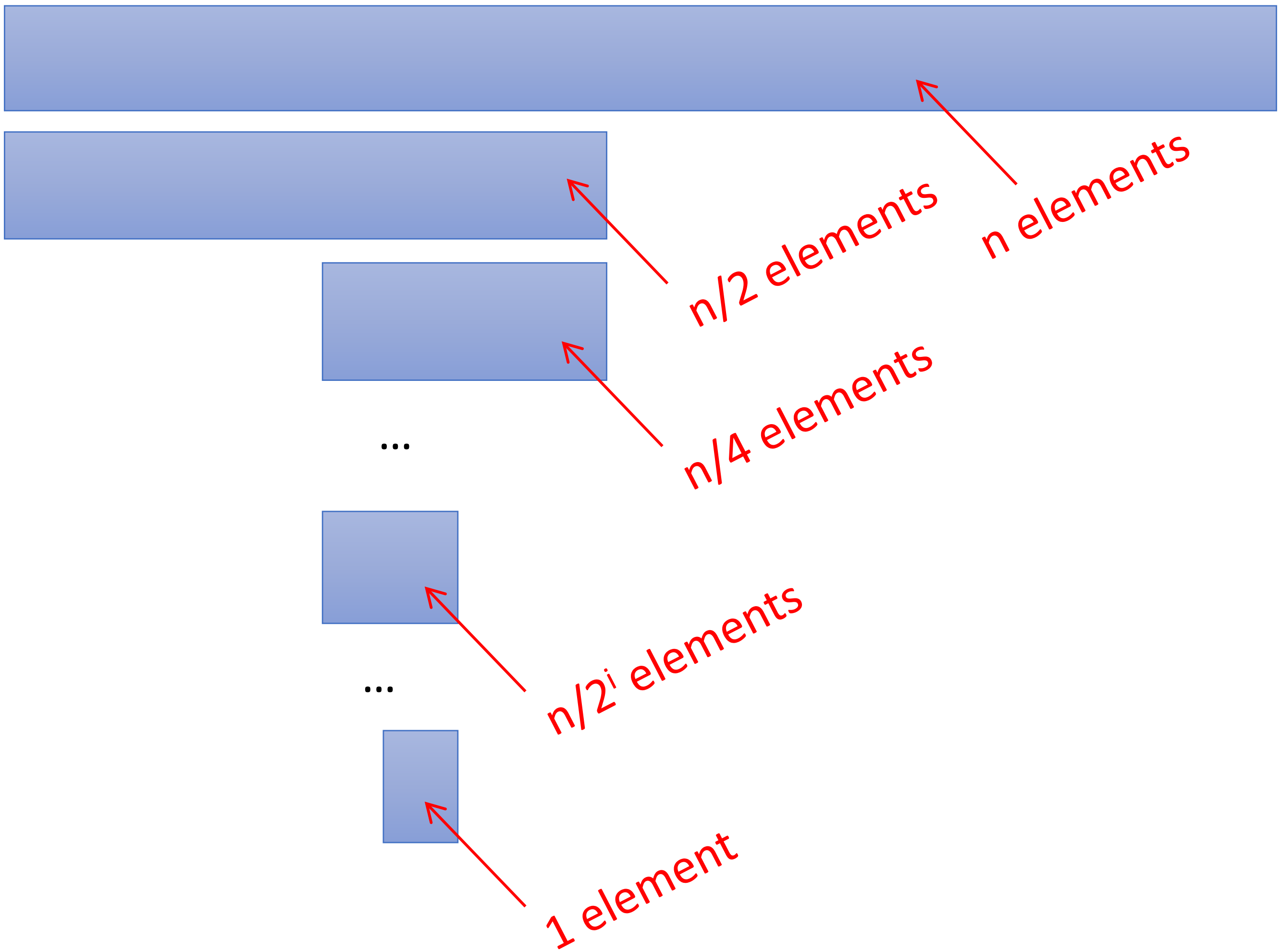
- Must only look until reach a number greater than e
- **Θ(len(L))** for the loop * **Θ(1)** to test if i == e or i > e
- Overall complexity is **Θ(len(L))**
  Θ(n) where n is len(L)

BISECTION SEARCH FOR AN ELEMENT IN A SORTED LIST

1) Pick an index, `i`, that divides list in half
2) Ask if `L[i] == e`
3) If not, ask if `L[i]` is larger or smaller than `e`
4) Depending on answer, search left or right half of `L` for `e`

- A new version of **divide-and-conquer: recursion!**
- Break into smaller versions of problem (smaller list), plus simple operations
- Answer to smaller version is answer to original version

BISECTION SEARCH COMPLEXITY ANALYSIS



- Finish looking through list when

  $1 = n/2^i$

- So… relationship between original length of list and how many times we divide the list: i = log n

- Complexity is **Θ(log n) where n is len(L)**

BIG IDEA

Two different implementations have two different Θ values.

BISECTION SEARCH IMPLEMENTATION 1

```
def bisect_search1(L, e):
    if L == []:
        return False
    elif len(L) == 1:
        return L[0] == e
    else:
        half = len(L)//2
        if L[half] > e:
            return bisect_search1( L[:half], e)
        else:
            return bisect_search1( L[half:], e)
```

constant
Θ(1)

constant
Θ(1)

constant
Θ(1)

NOT constant, copies list with each function call

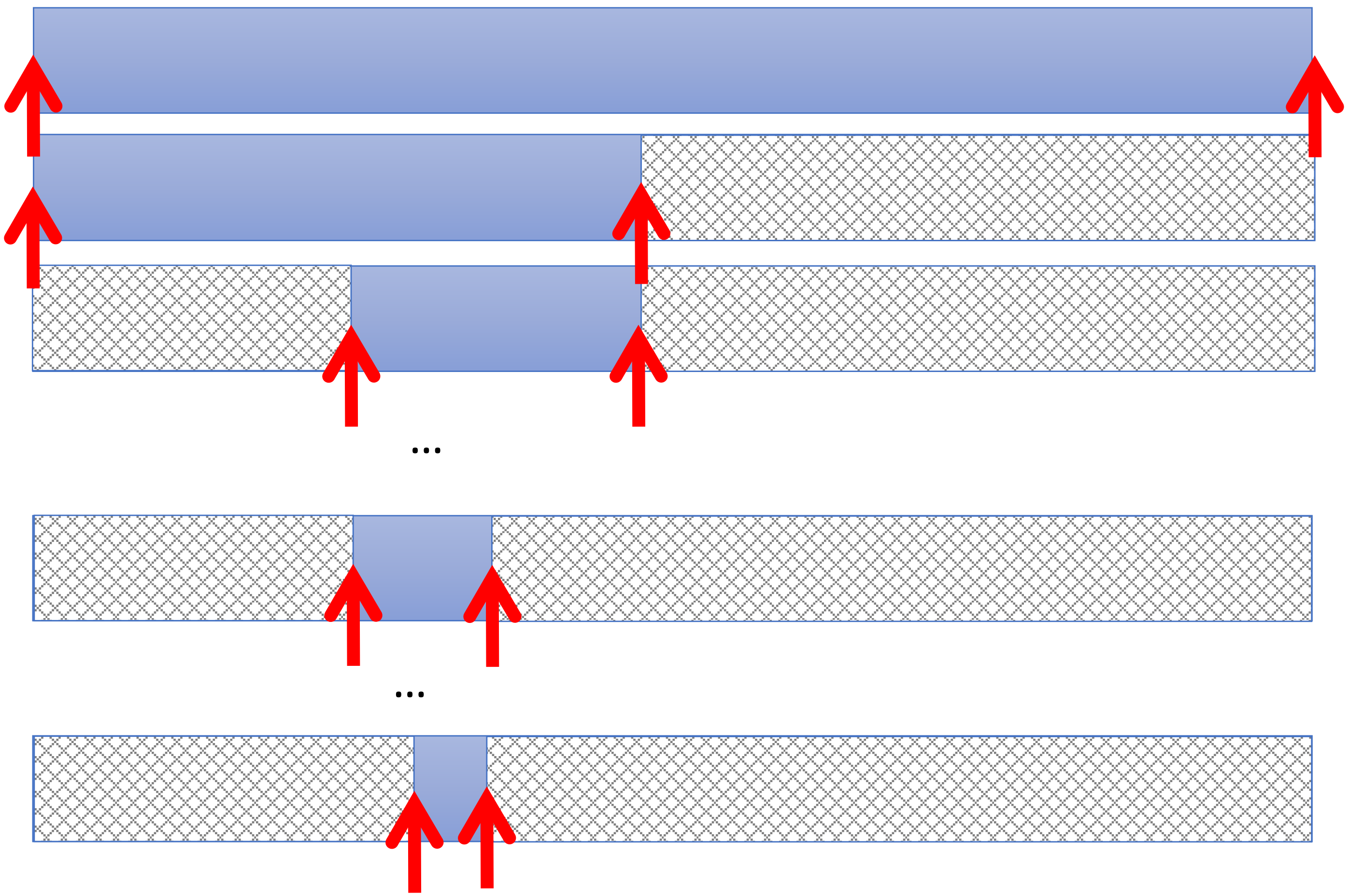
NOT constant
Θ(log(len(L)))

NOT constant
Θ(log(len(L)))

COMPLEXITY OF bisect_search1 (where n is len(L))

- **Θ(log n)** bisection search calls
  - Each recursive call cuts range to search in half
  - Worst case to reach range of size 1 from n is when $n/2^k = 1$ or when $k = \log n$
  - We do this to get an expression relating k to n
- **Θ(n)** for each bisection search call to copy list
  - Cost to set up recursive call at each level of recursion
- Θ(log n) * Θ(n) = **Θ(n log n) where n = len(L)**
  ^ this is the answer in this class
- If careful, notice list is also halved on each recursive call
  - Infinite series (don't worry about this in this class)
  - Θ(n) is a tighter bound because copying list dominates log n

BISECTION SEARCH ALTERNATE IMPLEMENTATION



- Reduce size of problem by factor of 2 each step
- Keep track of low and high indices to search list
- Avoid copying list
- Complexity of recursion is **Θ(log n) where n is len(L)**

BISECTION SEARCH IMPLEMENTATION 2

Instead of copying the list, keep track of the low and high list indices

```
def bisect_search2(L, e):
    def bisect_search_helper(L, e, low, high):
        if high == low:
            return L[low] == e
        mid = (low + high)//2
        if L[mid] == e:
            return True
        elif L[mid] > e:
            if low == mid: #nothing left to search
                return False
            else:
                return bisect_search_helper(L, e, low, mid - 1)
        else:
            return bisect_search_helper(L, e, mid + 1, high)
    if len(L) == 0:
        return False
    else:
        return bisect_search_helper(L, e, 0, len(L) - 1)
```

NOT constant
Θ(log(len(L))

NOT constant
Θ(log(len(L))

Kick off the recursive helper

COMPLEXITY OF bisect_search2 and helper (where n is len(L))

- **Θ(log n)** bisection search calls
  - Each recursive call cuts range to search in half
  - Worst case to reach range of size 1 from n is when $n/2^k = 1$ or when $k = \log n$
  - We do this to get an expression relating k to n
- Pass list and indices as parameters
  - List never copied, just re-passed
  - **Θ(1)** on each recursive call
- Θ (log n) * Θ(1) = **Θ(log n) where n is len(L)**

WHEN TO SORT FIRST AND THEN SEARCH?

SEARCHING A SORTED LIST -- n is len(L)

- Using **linear search**, search for an element is **Θ(n)**
- Using **binary search**, can search for an element in **Θ(log n)**
  - Assumes the **list is sorted**!
- When does it make sense to **sort first then search**?

  - $\overbrace{\boxed{\text{SORT}}}^{\text{Time to sort}} + \overbrace{\boxed{\Theta(\log n)}}^{\text{Time for binary search}} < \overbrace{\boxed{\Theta(n)}}^{\text{Time for linear search}}$
    implies that SORT < $\Theta(n) - \Theta(\log n)$

  - When is sorting is less than $\Theta(n)$??!!?
    → Never true because you'd at least have to look at each element!

AMORTIZED COST
-- n is len(L)

- Why bother sorting first?
- **Sort a list once** then do **many searches**

- **AMORTIZE cost** of the sort over many searches

Only once!

Do K searches

- $\boxed{\text{SORT}} + \boxed{\texttt{K}} * \Theta(\texttt{log n}) < \boxed{\texttt{K}} * \Theta(\texttt{n})$

  implies that for large K, **SORT time becomes irrelevant**

COMPLEXITY CLASSES SUMMARY

- Compare efficiency of algorithms
- Lower order of growth
- Using **Θ for an upper and lower ("tight") bound**

- Given a function f:
  - Only look at **items in terms of the input**
  - Look at **loops**
    - Are they in terms of the input to f?
    - Are there nested loops?
  - Look at **recursive calls**
    - How deep does the function call stack go?
  - Look at **built-in functions**
    - Any of them depend on the input?

6.100L Introduction to Computer Science and Programming Using Python
Fall 2022

SORTING ALGORITHMS

(download slides and .py files to follow along)

6.100L Lecture 24

Ana Bell

SEARCHING A SORTED LIST
-- n is len(L)

- Using **linear search**, search for an element is **Θ(n)**
- Using **binary search**, can search for an element in **Θ(logn)**
  - assumes the **list is sorted**!
- When does it make sense to **sort first then search**?

Time to sort

Time for binary search

Time for linear search

$\boxed{\text{SORT}} + \boxed{\Theta(\log n)} < \boxed{\Theta(n)}$ implies SORT $< \Theta(n) - \Theta(\log n)$

When sorting is less than $\Theta(n)$!?!? This is never true!

AMORTIZED COST
-- n is len(L)

- Why bother sorting first?
- **Sort a list once** then do **many searches**
- **AMORTIZE cost** of the sort over many searches

Only once!

Do K searches

- $\boxed{\text{SORT}} + \boxed{\texttt{K}} * \Theta(\texttt{log n}) < \boxed{\texttt{K}} * \Theta(\texttt{n})$

  → for large `K`, **SORT time becomes irrelevant**

SORTING ALGORITHMS

BOGO/RANDOM/MONKEY SORT

- aka bogosort, stupidsort, slowsort, randomsort, shotgunsort
- To sort a deck of cards
  - throw them in the air
  - pick them up
  - are they sorted?
  - repeat if not sorted

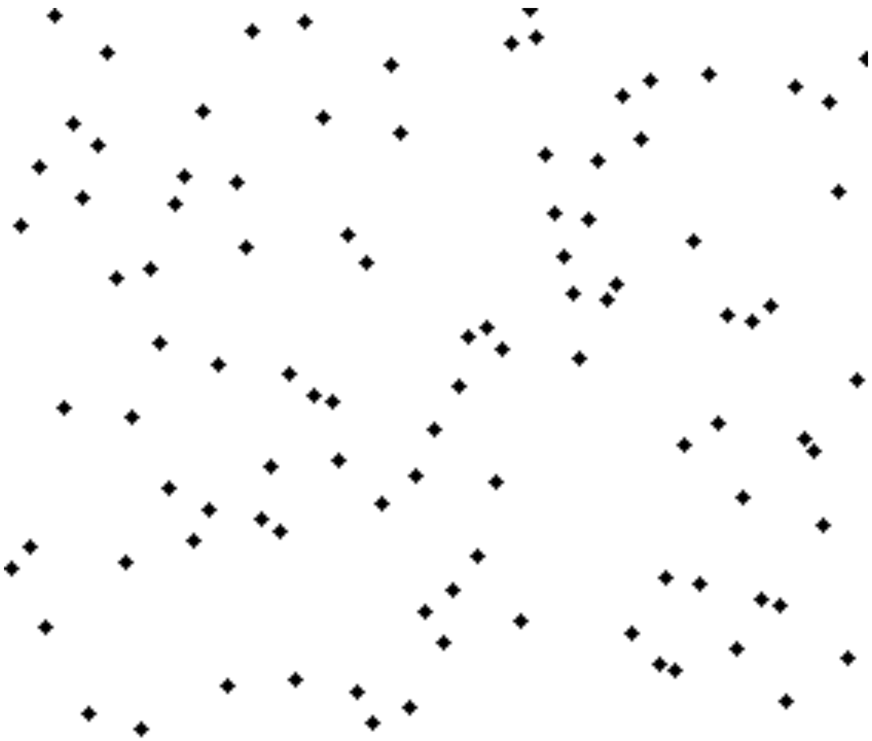
COMPLEXITY OF BOGO SORT

```
def bogo_sort(L):
    while not is_sorted(L):
        random.shuffle(L)
```

- Best case: **Θ(n) where n is len(L)** to check if sorted
- Worst case: Θ(?) it is **unbounded** if really unlucky

BUBBLE SORT

- **Compare consecutive pairs** of elements
- **Swap elements** in pair such that smaller is first
- When reach end of list, **start over** again
- Stop when **no more swaps** have been made



Donald Knuth, in "The Art of Computer Programming", said:
"the bubble sort seems to have nothing to recommend it, except a catchy name and the fact that it leads to some interesting theoretical problems"

COMPLEXITY OF BUBBLE SORT

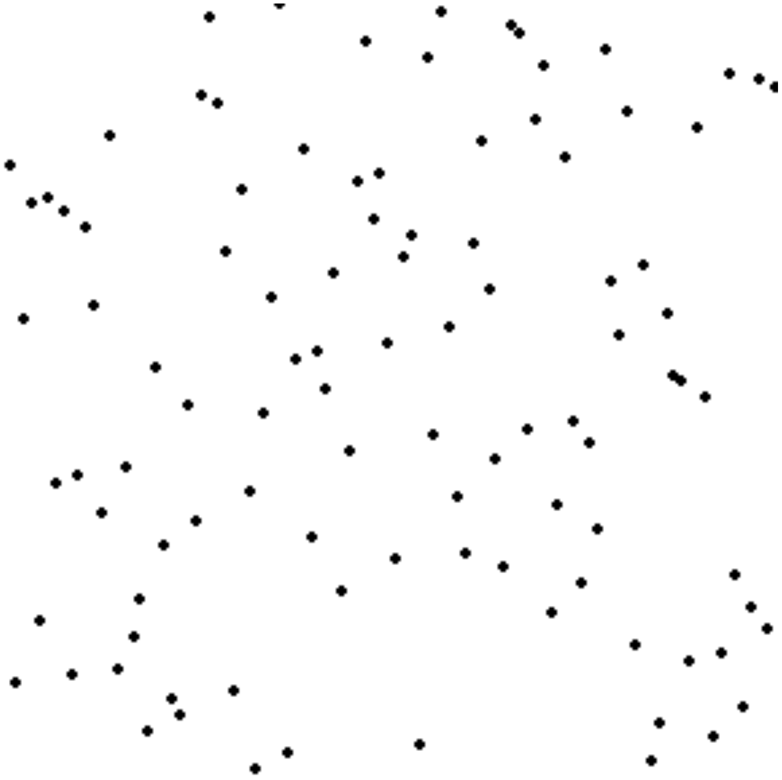
```
def bubble_sort(L):
    did_swap = True
    while did_swap:
        did_swap = False
        for j in range(1, len(L)):
            if L[j-1] > L[j]:
                did_swap = True
                L[j],L[j-1] = L[j-1],L[j]
```

Θ(len(L))

Θ(len(L))

- Inner for loop is for doing the **comparisons**
- Outer while loop is for doing **multiple passes** until no more swaps
- **$\Theta(n^2)$ where n is len(L)**
  to do len(L)-1 comparisons and len(L)-1 passes

SELECTION SORT



- First step
  - Extract **minimum element**
  - **Swap it** with element at **index 0**
- Second step
  - In remaining sublist, extract **minimum element**
  - **Swap it** with the element at **index 1**
- Keep the left portion of the list sorted
  - At ith step, **first i elements in list are sorted**
  - All other elements are bigger than first i elements

COMPLEXITY OF SELECTION SORT

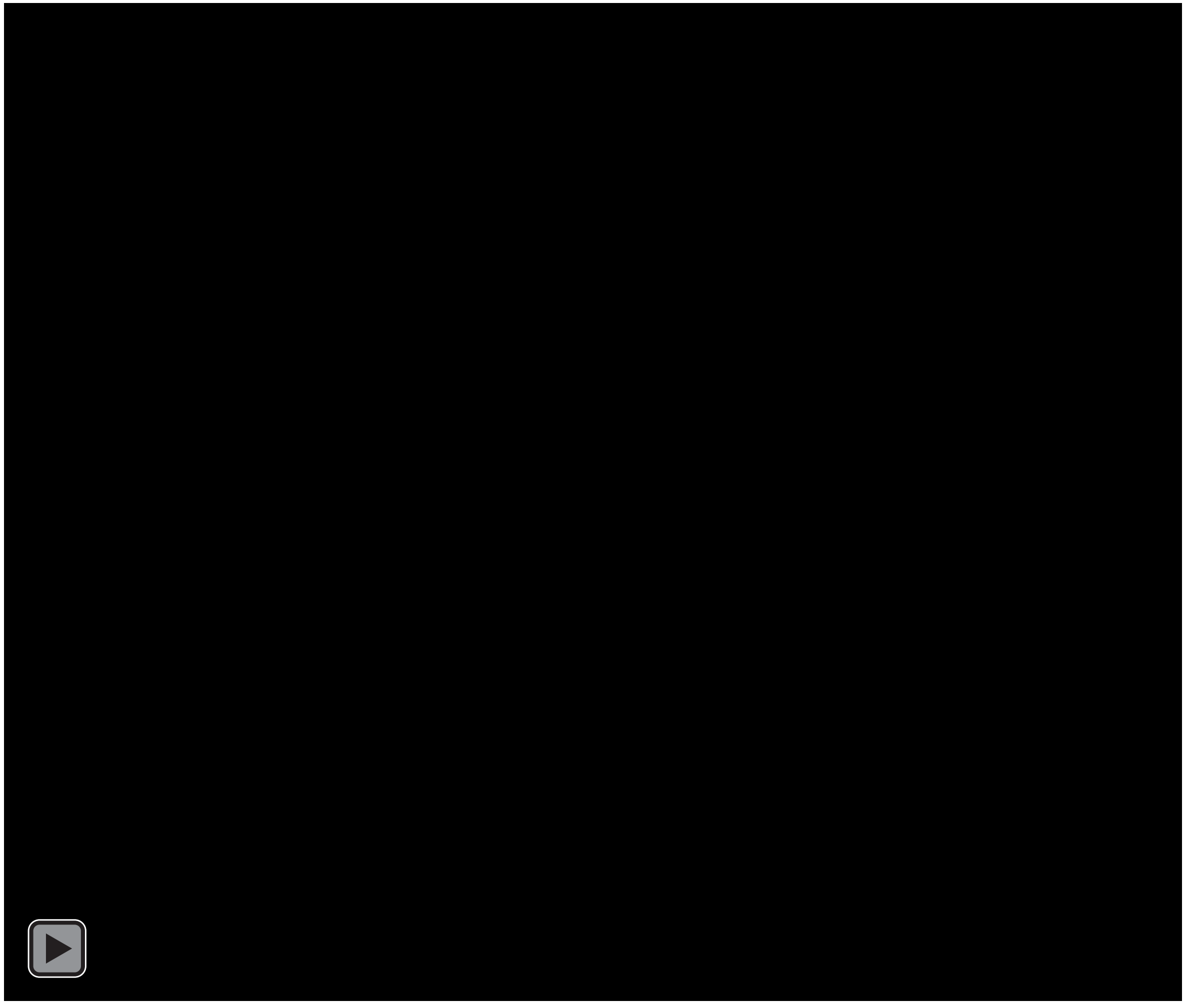
```
def selection_sort(L):
    for i in range(len(L)):
        for j in range(i, len(L)):
            if L[j] < L[i]:
                L[i], L[j] = L[j], L[i]
```

len(L) times
→ Θ(len(L))

len(L) – i times
→ Θ(len(L))

- Complexity of selection sort is **$\Theta(n^2)$ where n is len(L)**
  - Outer loop executes len(L) times
  - Inner loop executes len(L) – i times, on avg len(L)/2
- Can also think about how many times the comparison happens over both loops: say n = len(L)
  - Approx $1+2+3+\ldots+n = (n)(n+1)/2 = n^2/2+n/2 = \Theta(n^2)$

VARIATION ON SELECTION SORT: don’t swap every time

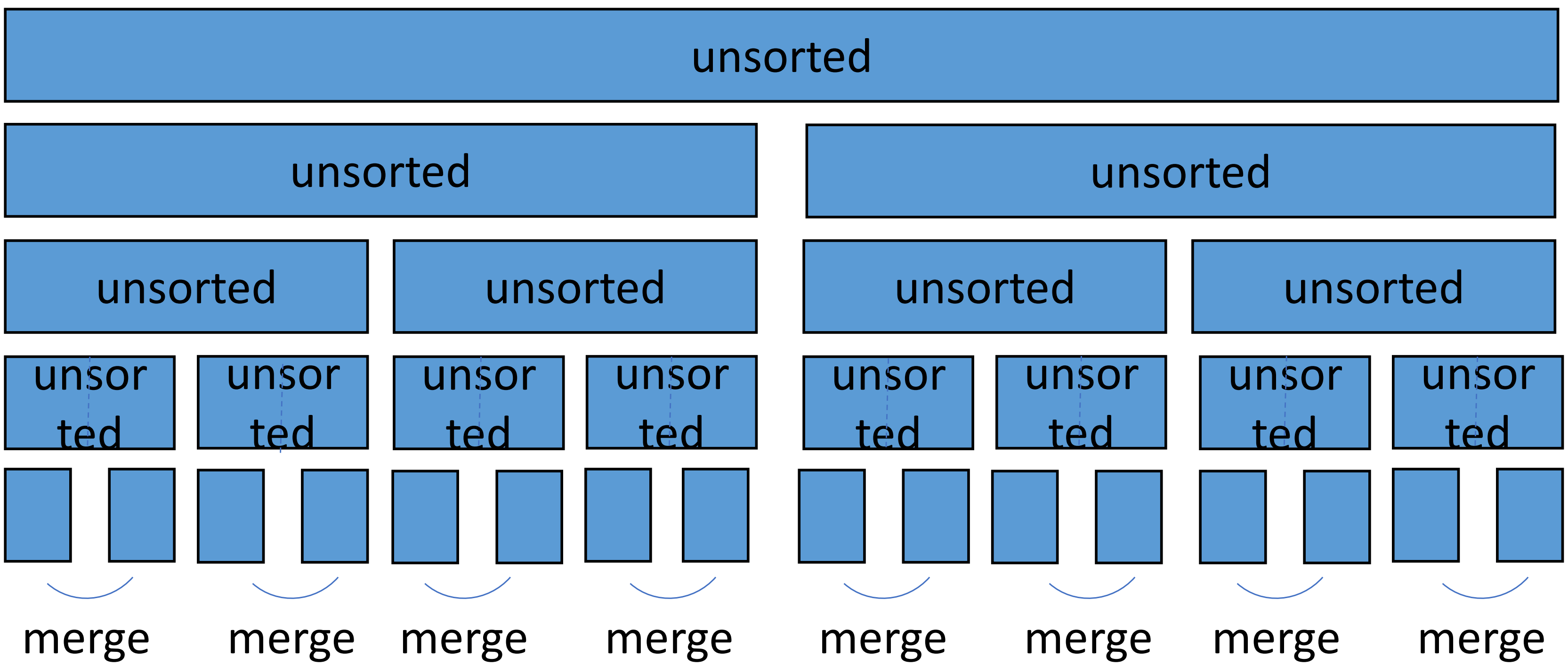


MERGE SORT

- Use a **divide-and-conquer** approach:
  - If list is of length 0 or 1, already sorted
  - If list has more than one element, split into two lists, and sort each
  - Merge sorted sublists
    - Look at first element of each, move smaller to end of the result
    - When one list empty, just copy rest of other list

MERGE SORT

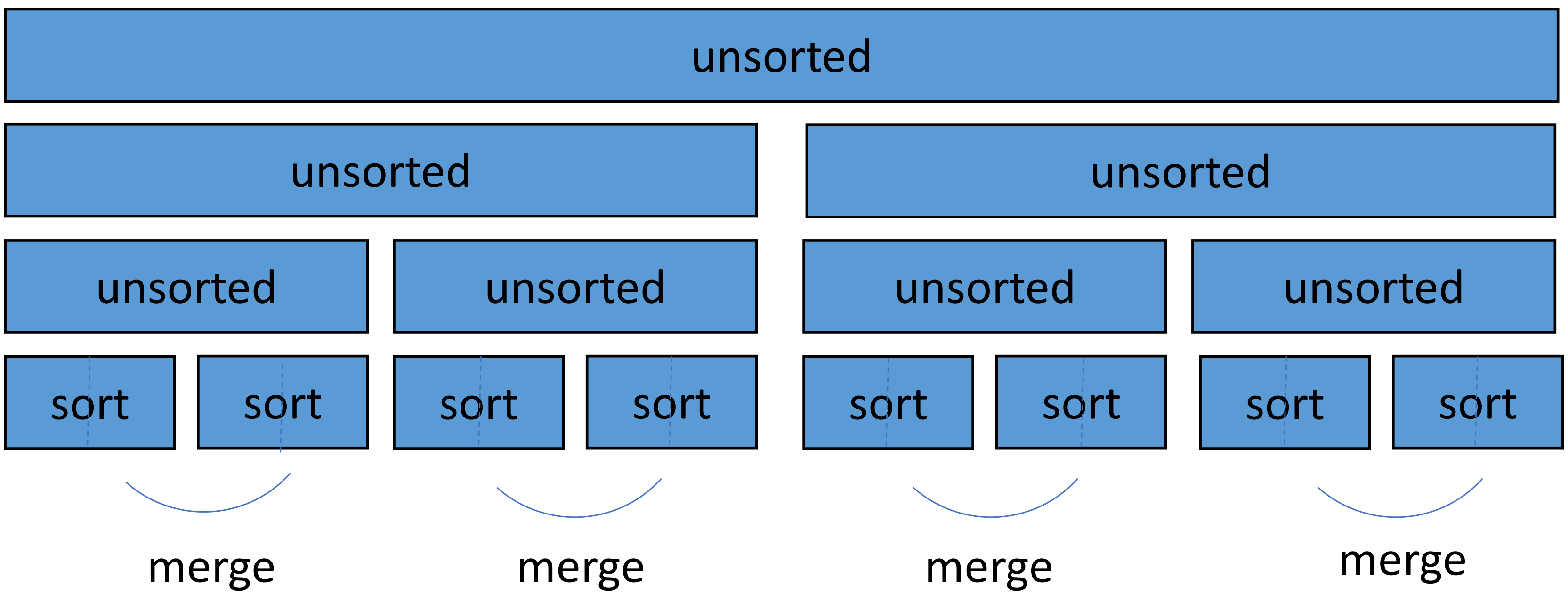
- Divide and conquer



- **Split list in half** until have sublists of only 1 element

MERGE SORT

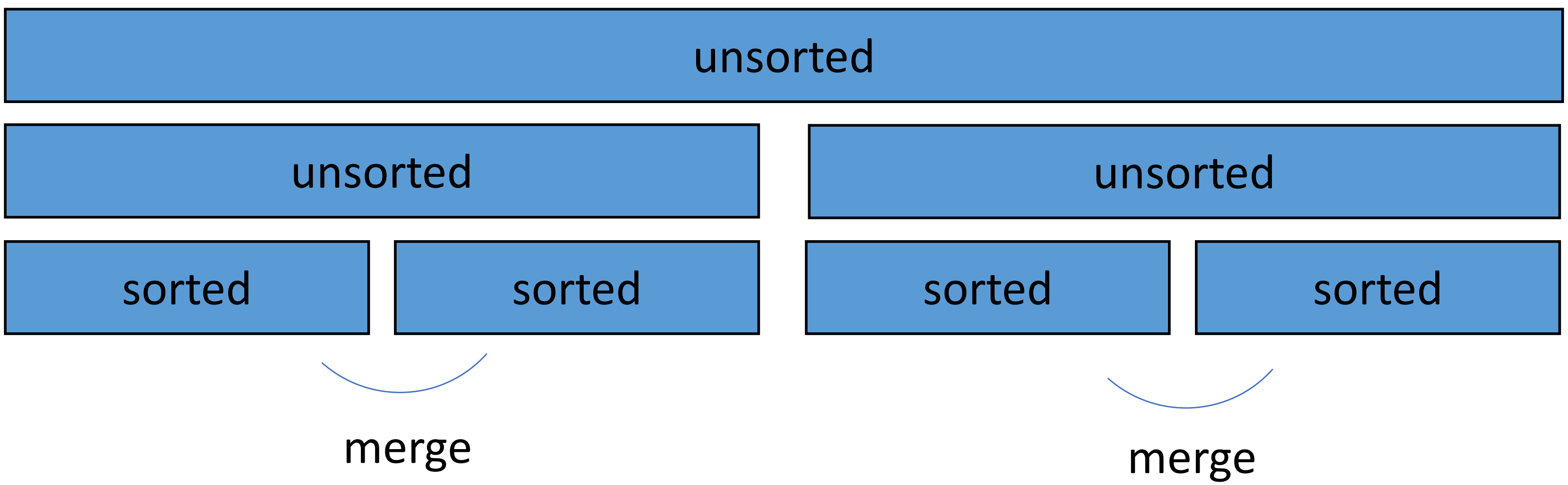
- Divide and conquer



- Merge such that **sublists will be sorted after merge**

MERGE SORT

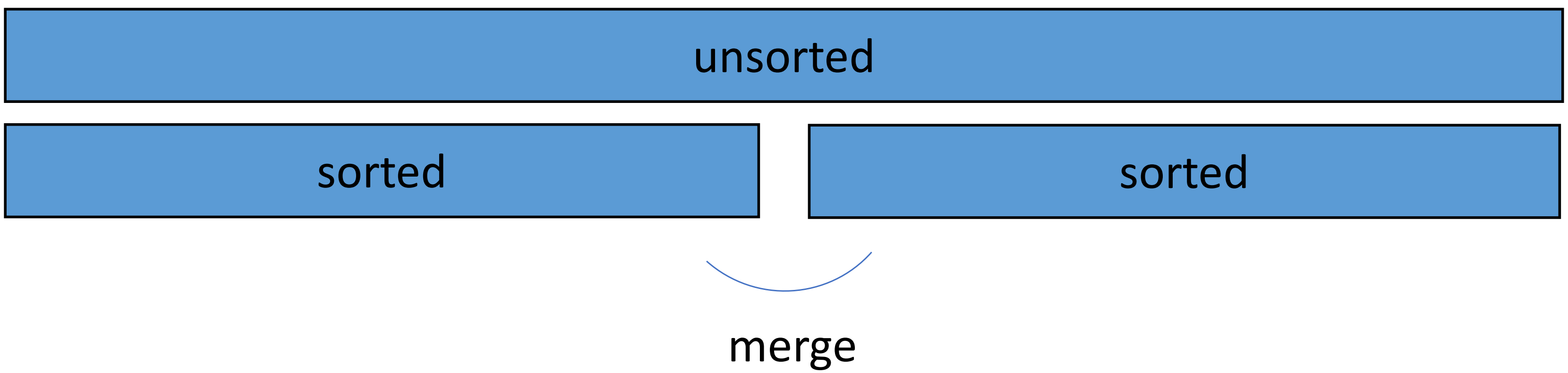
- Divide and conquer



- Merge sorted sublists
- Sublists will be sorted after merge

MERGE SORT

- Divide and conquer



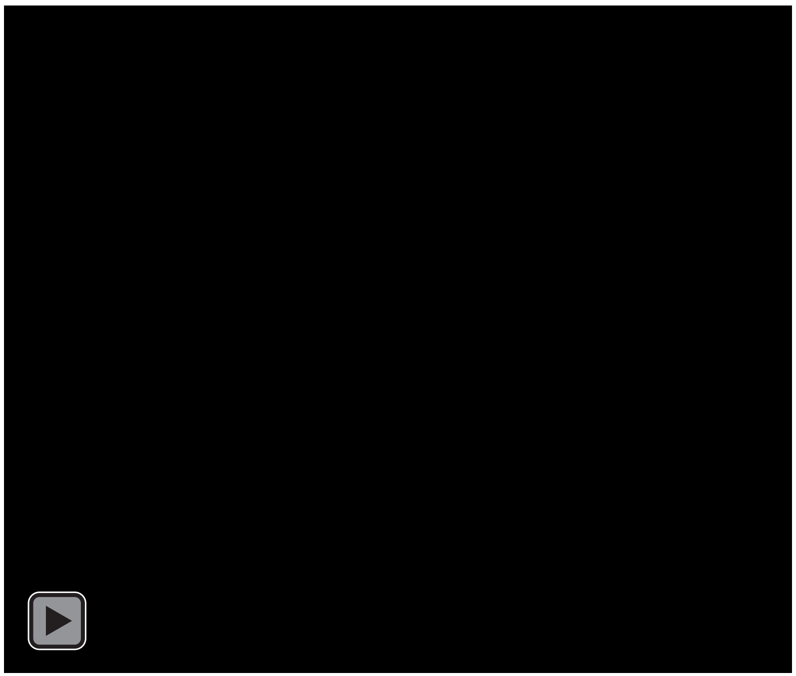
- Merge sorted sublists
- Sublists will be sorted after merge

MERGE SORT

- Divide and conquer – done!

sorted

MERGE SORT DEMO



1. Recursively divide into subproblems
2. Sort each subproblem using linear merge
3. Merge (sorted) subproblems into output list

CLOSER LOOK AT THE MERGE STEP (EXAMPLE)

Left in list 1	Left in list 2	Compare	Result
[1,5,12,18,19,20]	[2,3,4,17]	1, 2	[]
[5,12,18,19,20]	[2,3,4,17]	5, 2	[1]
[5,12,18,19,20]	[3,4,17]	5, 3	[1,2]
[5,12,18,19,20]	[4,17]	5, 4	[1,2,3]
[5,12,18,19,20]	[17]	5, 17	[1,2,3,4]
[12,18,19,20]	[17]	12, 17	[1,2,3,4,5]
[18,19,20]	[17]	18, 17	[1,2,3,4,5,12]
[18,19,20]	[]	18, --	[1,2,3,4,5,12,17]
[]	[]		

[1,2,3,4,5,12,17,18,19,20]

MERGING SUBLISTS STEP

```
def merge(left, right):
    result = []
    i,j = 0, 0
    while i < len(left) and j < len(right):
        if left[i] < right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1
    while (i < len(left)):
        result.append(left[i])
        i += 1
    while (j < len(right)):
        result.append(right[j])
        j += 1
    return result
```

- Left and right sublists are ordered
- Move indices for sublists depending on which sublist holds next smallest element

When right sublist is empty

When left sublist is empty

COMPLEXITY OF MERGING STEP

- Go through two lists, only one pass
- Compare only **smallest elements in each sublist**
- Θ(len(left) + len(right)) copied elements
- Worst case Θ(len(longer list)) comparisons
- **Linear in length of the lists**

FULL MERGE SORT ALGORITHM
-- RECURSIVE

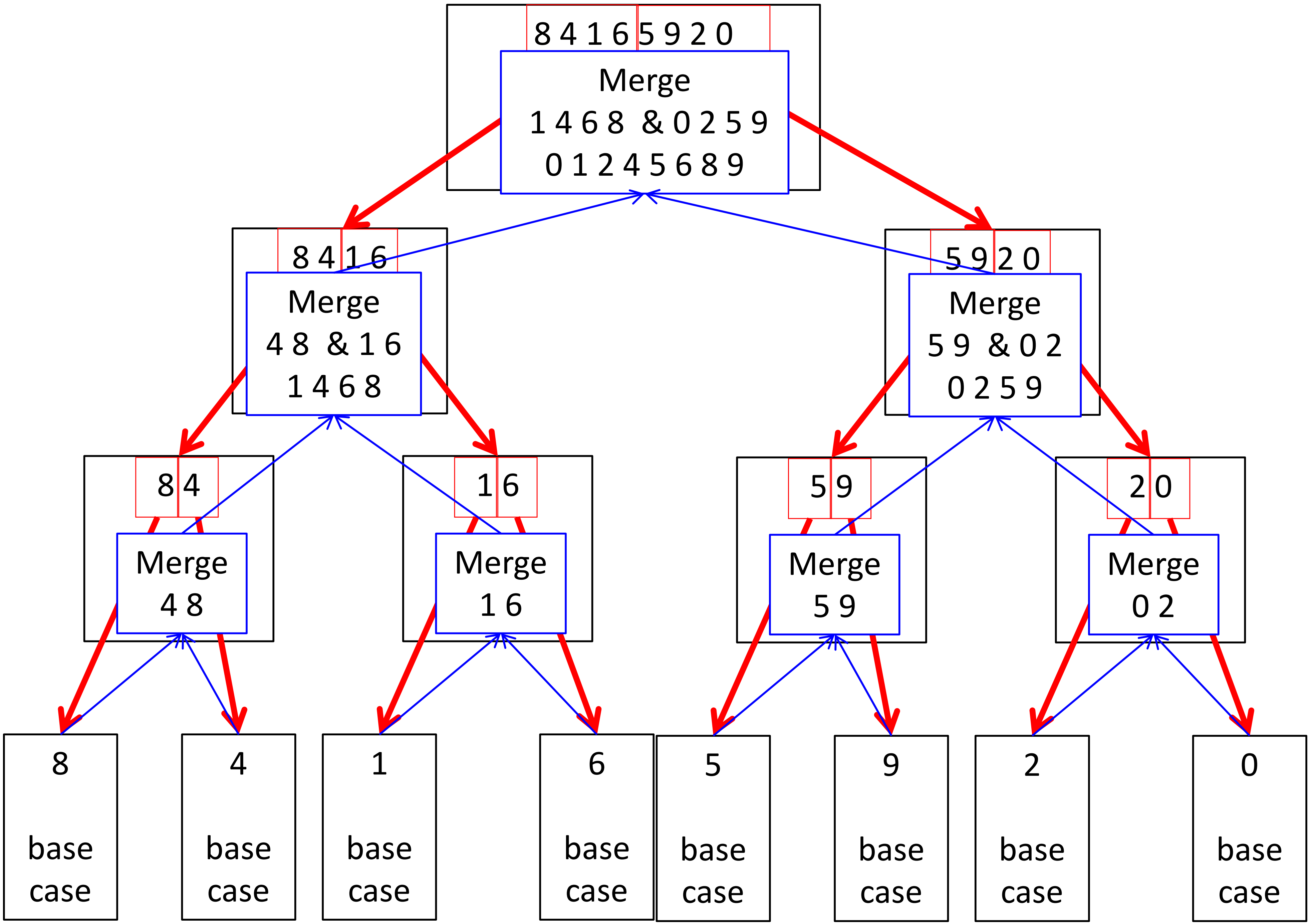
```
def merge_sort(L):
    if len(L) < 2:
        return L[:]
    else:
        middle = len(L)//2
        left = merge_sort(L[:middle])
        right = merge_sort(L[middle:])
        return merge(left, right)
```

base case

divide

conquer with the merge step

- **Divide list** successively into halves
- Depth-first such that **conquer smallest pieces down one branch** first before moving to larger pieces


8 4 1 6 5 9 2 0
Merge
1 4 6 8 & 0 2 5 9
0 1 2 4 5 6 8 9
8 4 1 6
Merge
4 8 & 1 6
1 4 6 8
5 9 2 0
Merge
5 9 & 0 2
0 2 5 9
8 4
Merge
4 8
1 6
Merge
1 6
5 9
Merge
5 9
2 0
Merge
0 2
8 base case
4 base case
1 base case
6 base case
5 base case
9 base case
2 base case
0 base case

COMPLEXITY OF MERGE SORT

- Each level
  - At **first recursion level**
    - n/2 elements in each list, 2 lists
    - One merge → $\Theta(n) + \Theta(n) = \Theta(n)$ where n is len(L)
  - At **second recursion level**
    - n/4 elements in each list, 4 lists
    - Two merges → $\Theta(n)$ where n is len(L)
  - And so on…
- **Dividing list in half** with each recursive call gives our levels
  - $\Theta(\log n)$ where n is len(L)
  - Like bisection search: $1 = n/2^i$ tells us how many splits to get to one element
- Each recursion level does $\Theta(n)$ work and there are $\Theta(\log n)$ levels, where n is len(L)
- Overall complexity is **$\Theta(n \log n)$ where n is len(L)**

SORTING SUMMARY
-- n is len(L)

- Bogo sort
  - Randomness, unbounded $\Theta()$
- Bubble sort
  - $\Theta(n^2)$
- Selection sort
  - $\Theta(n^2)$
  - Guaranteed the first i elements were sorted
- Merge sort
  - $\Theta(n \log n)$
- **$\Theta(n \log n)$ is the fastest a sort can be**

COMPLEXITY SUMMARY

- Compare **efficiency of algorithms**
  - Describe **asymptotic** order of growth with Big Theta
  - **Worst case** analysis
- Saw different classes of complexity
  - Constant
  - Log
  - Linear
  - Log linear
  - Polynomial
  - Exponential
- A priori evaluation (before writing or running code)
- Assesses algorithm independently of machine and implementation
- Provides direct insight to the **design** of efficient algorithms

PLOTTING

(download slides and .py files to follow along)

6.100L Lecture 25

Ana Bell

WHY PLOTTING?

- Sooner or later, everyone needs to produce plots
  - Helps us **visualize** data to see trends, pose **computational questions** to probe
  - If you join 6.100B, you will make extensive use of them
  - For those of you leaving us after next week, this is a valuable way to visualize data
- Example of **leveraging an existing library**, rather than writing procedures from scratch
- Python provides libraries for:
  - Plotting
  - Numerical computation
  - Stochastic computation
  - Many others

MATPLOTLIB

- Can **import library** into computing environment

```
import matplotlib.pyplot as plt
```

  - Allows **code to reference library** procedures as `plt.<processName>`
- Provides access to existing set of graphing/plotting procedures
- Today will just show some simple examples; lots of additional information available in documentation associated with `matplotlib`
- Will see many other examples and details of these ideas if you take 6.100B

A SIMPLE EXAMPLE

- Idea – create different functions of a variable (n), and visualize their differences

```
nVals = []
linear = []
quadratic = []
cubic = []
exponential = []

for n in range(0, 30):
    nVals.append(n)
    linear.append(n)
    quadratic.append(n**2)
    cubic.append(n**3)
    exponential.append(1.5**n)
```

List of values of variable

Lists of values of functions of variable

Used 1.5 to keep displays visible, more common value for order of growth example would be 2

PLOTTING THE DATA

Typically n

Typically a function of n, e,g., f(n)

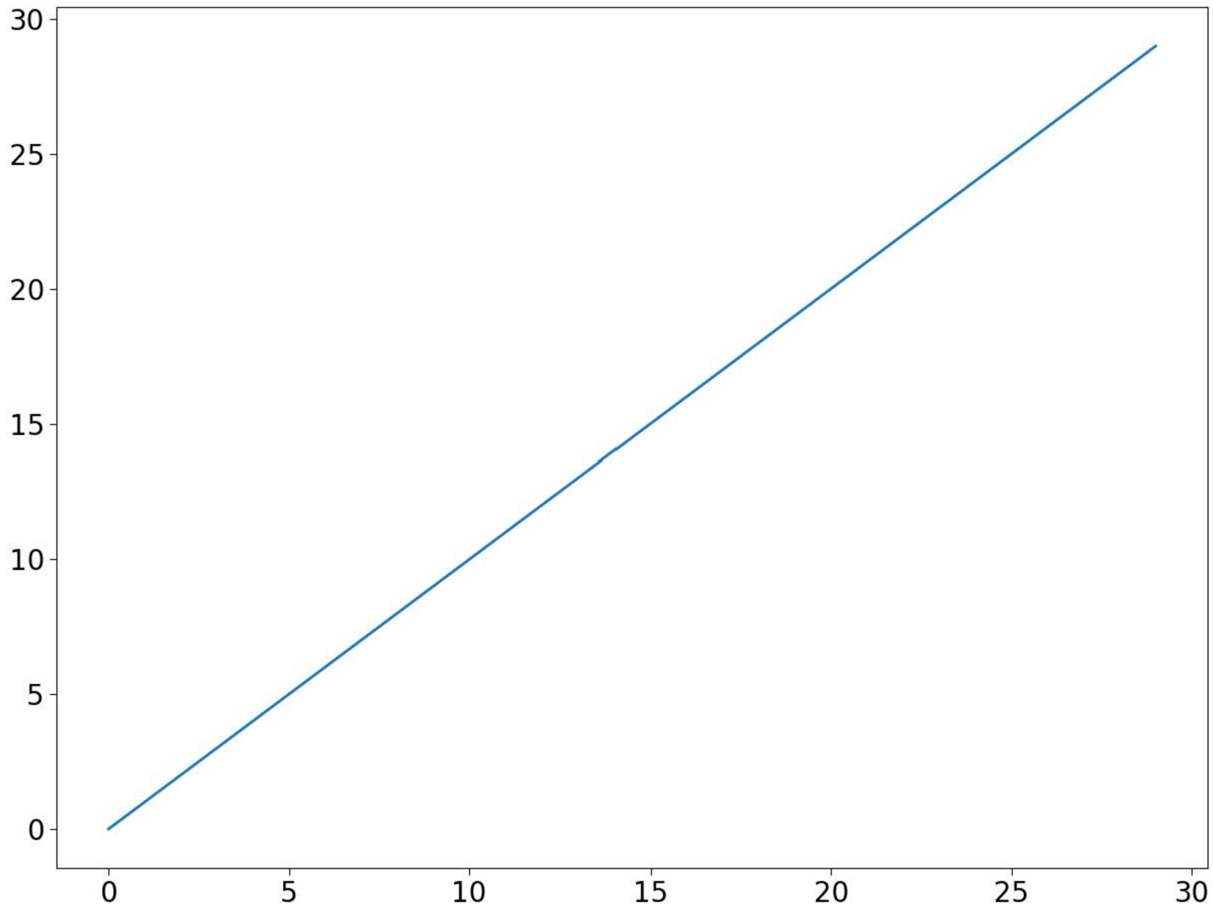
- To generate a plot:

```
plt.plot(<x values>, <y values>)
```

- Arguments are lists (or sequences) of numbers
  - Lists must be of the same length
  - Generates a sequence of <x, y> values on a Cartesian grid
  - Plotted in order, then connected with lines
- Can change iPython console to **generate plots in a new window** through Preferences
  - Inline in the console
  - In a new window

EXAMPLE

```
plt.plot(nVals, linear)
```



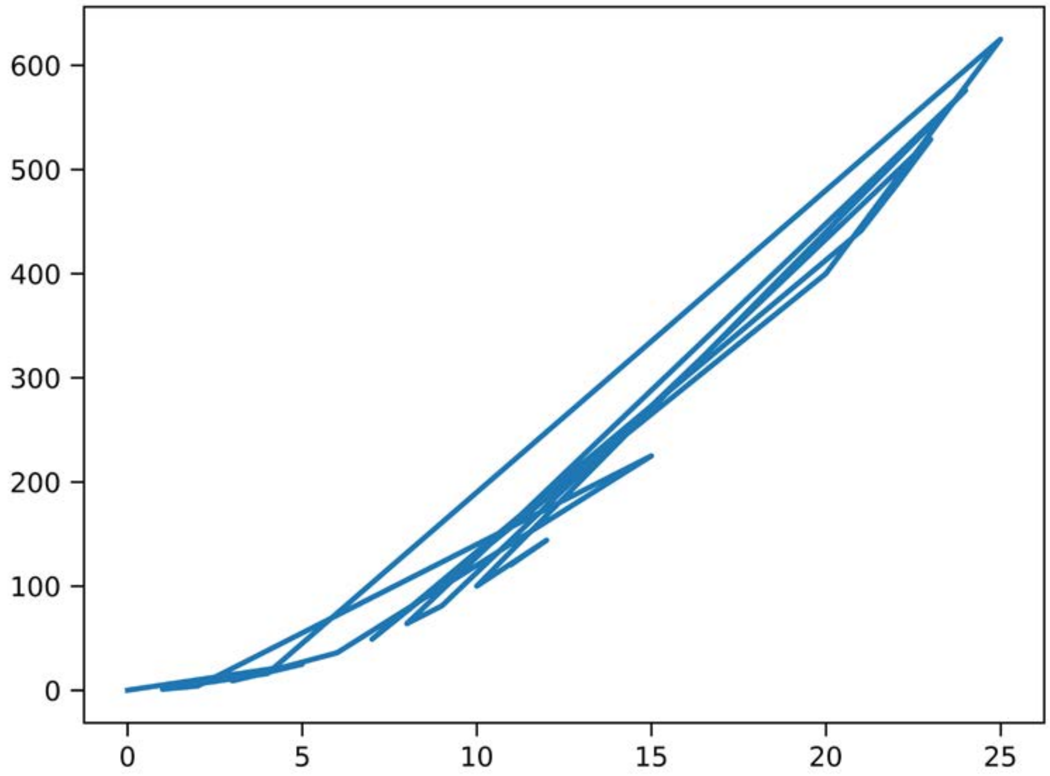
Note how `matplotlib` automatically fits plot within frame

ORDER OF POINTS MATTERS

- Suppose I create a set of values for n and for n^2, but in arbitrary order
- Python plots using the order of the points and connecting consecutive points

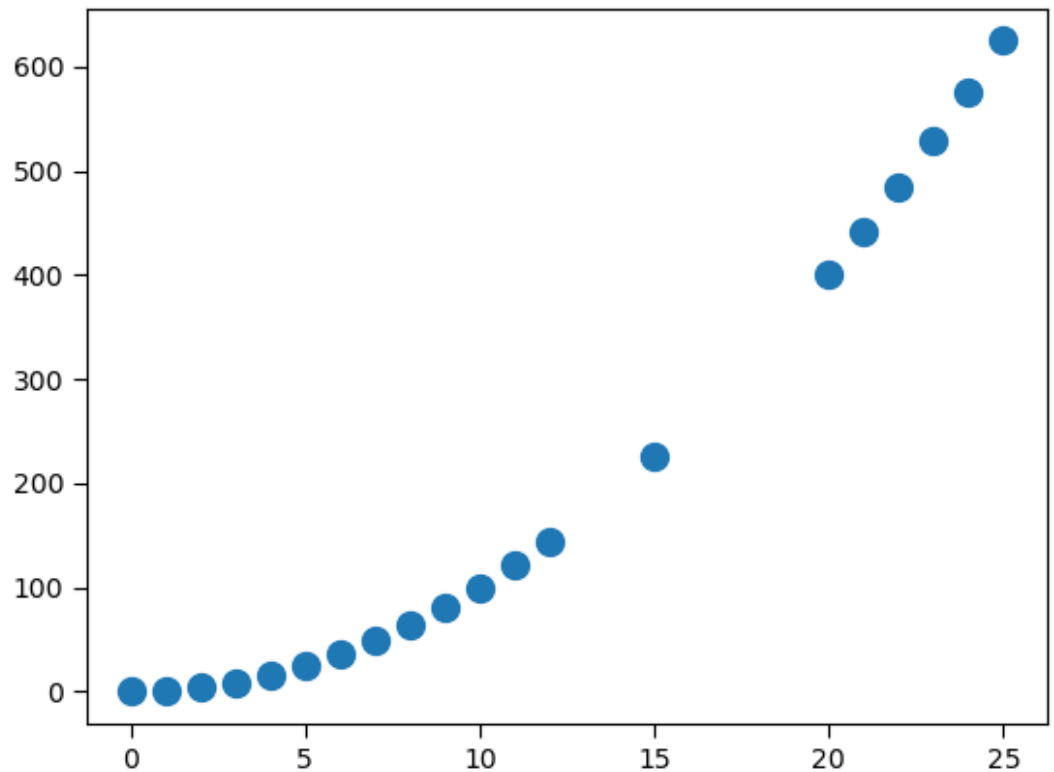
UNORDERED EXAMPLE

```
testSamples = [0,5,3,6,15,2,1,4,25,20,7,21,22,23,9,8,24,10,12,11]
testValues =  [0,25,9,36,225,4,1,16,625,400,49,441,484,529,81,64,576,100,144,121]
## plot connects the points
plt.plot(testSamples, testValues)
```



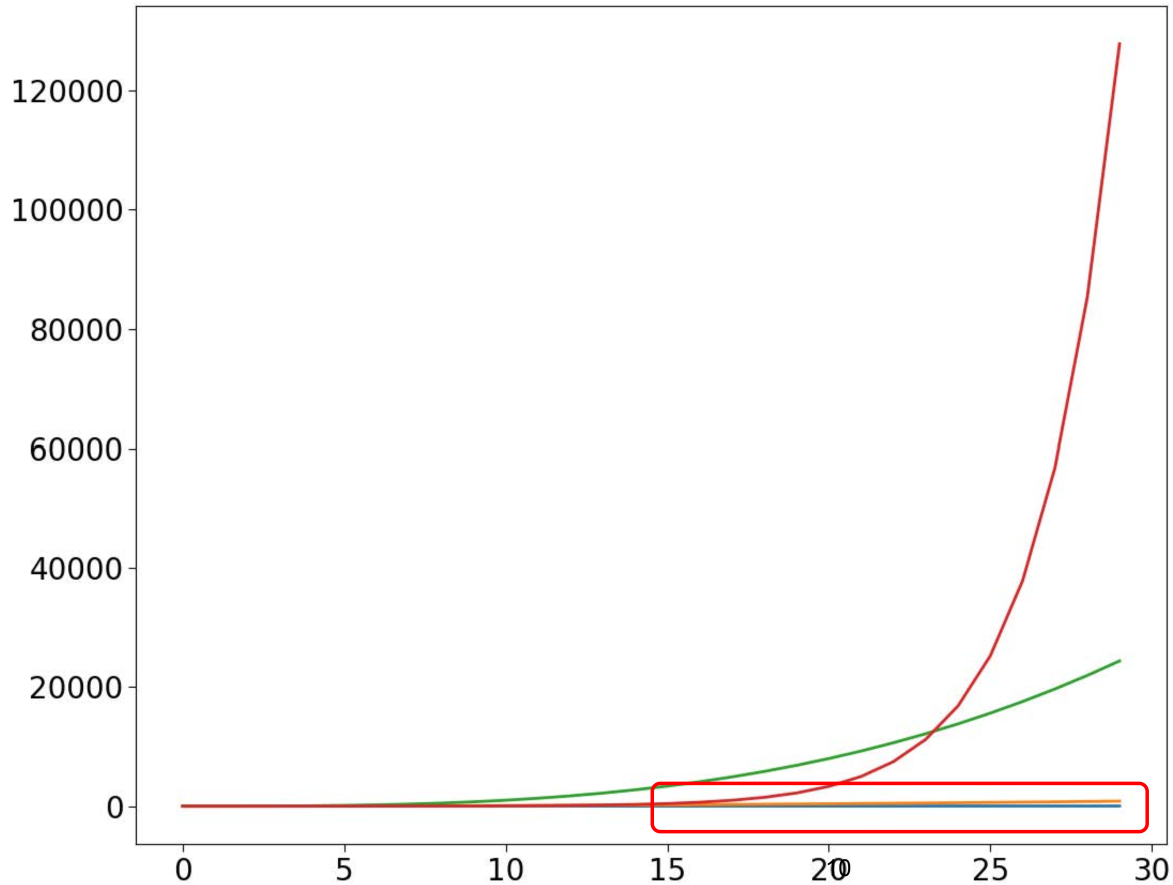
SCATTER PLOT DOES NOT CONNECT DATA POINTS

```
testSamples = [0,5,3,6,15,2,1,4,25,20,7,21,22,23,9,8,24,10,12,11]
testValues =  [0,25,9,36,225,4,1,16,625,400,49,441,484,529,81,64,576,100,144,121]
## scatter plot does not connect the points
plt.scatter(testSamples, testValues)
```



SHOWING ALL DATA ON ONE PLOT

```
plt.plot(nVals, linear)
plt.plot(nVals, quadratic)
plt.plot(nVals, cubic)
plt.plot(nVals, exponential)
```



Impossible to see linear graph, or even quadratic graph

Problem is that scales are very different

PRODUCING MULTIPLE PLOTS

- Let's graph each one in separate frame/window
- Call

```
plt.figure(<arg>)
```

gives a name to this figure; allows us to reference for future use

  - Creates a new display with that name if one does not already exist
  - If a display with that name exists, reopens it for additional processing

EXAMPLE CODE

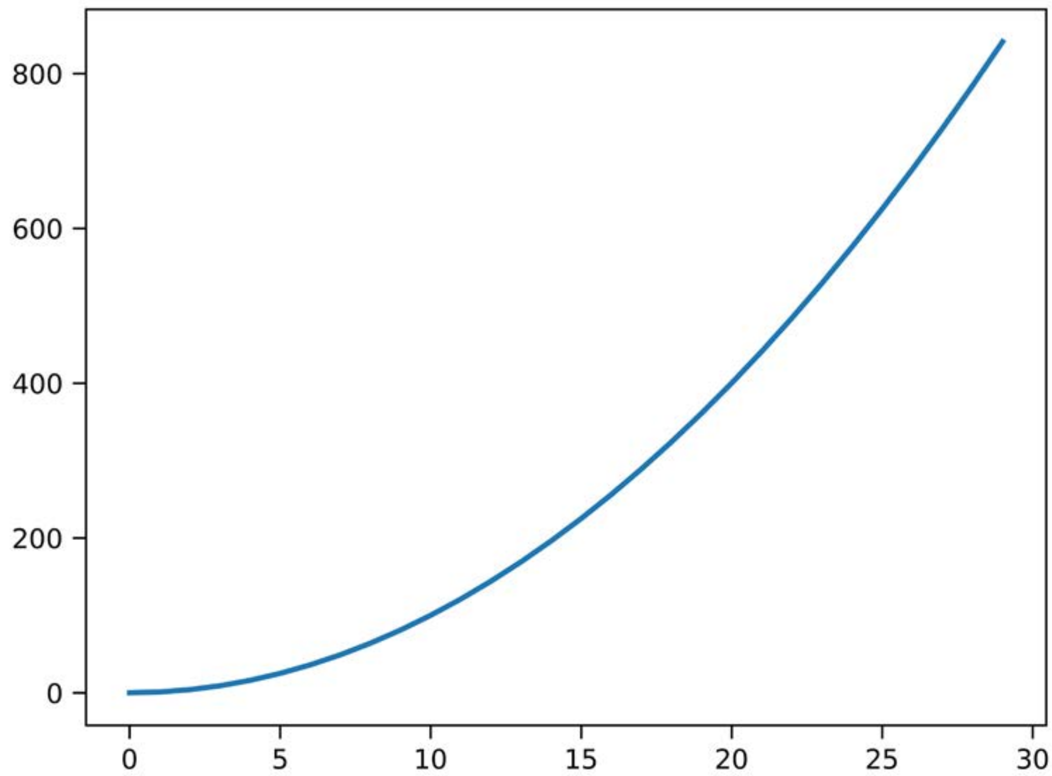
```
plt.figure('expo')
plt.plot(nVals, exponential)
plt.figure('lin')
plt.plot(nVals, linear)
plt.figure('quad')
plt.plot(nVals, quadratic)
plt.figure('cube')
plt.plot(nVals, cubic)
newExpo = []
for i in range(30):
    newExpo.append(1.6**i)
plt.figure('expo')
plt.plot(nVals, newExpo)
```

New figure with name expo
Plot inside that figure
New figure with name lin
Plot inside that figure

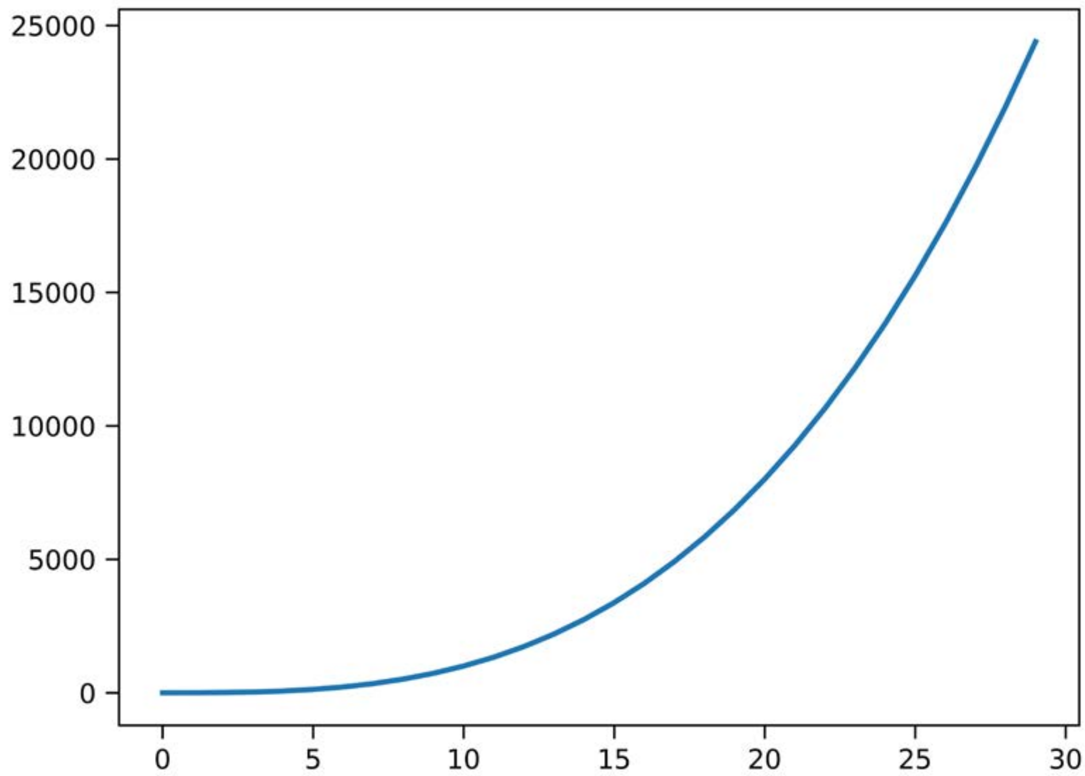
Make another exponential function

Go back to expo
Add another plot to that figure

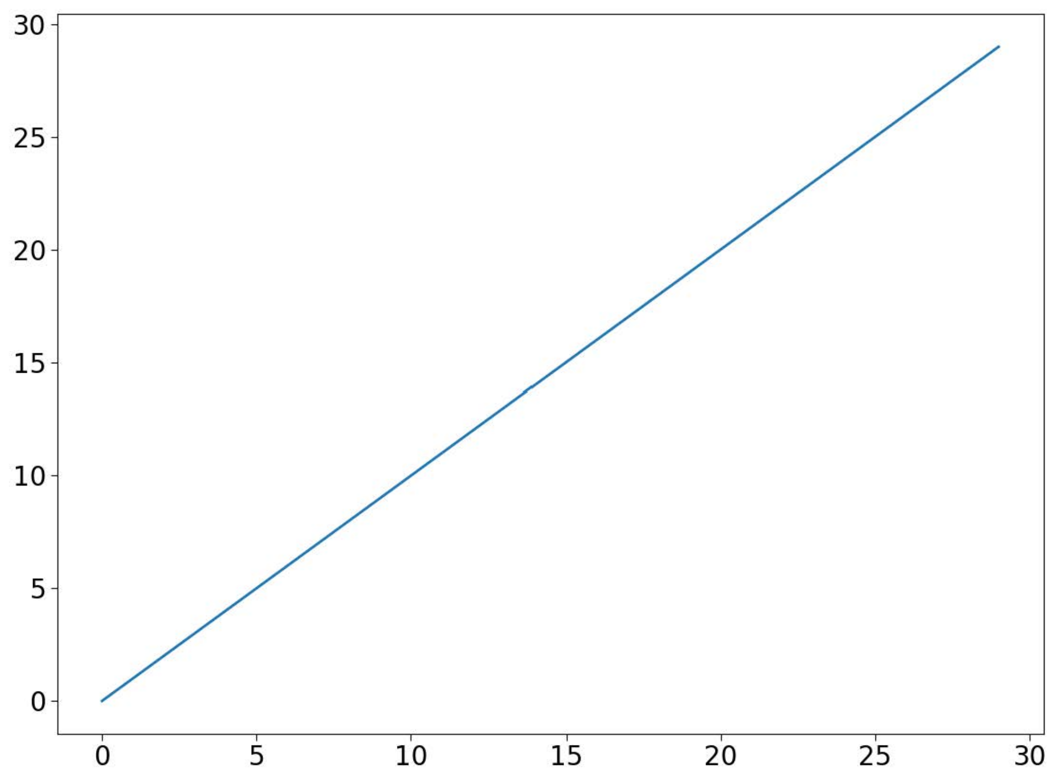
DISPLAY OF `quad`



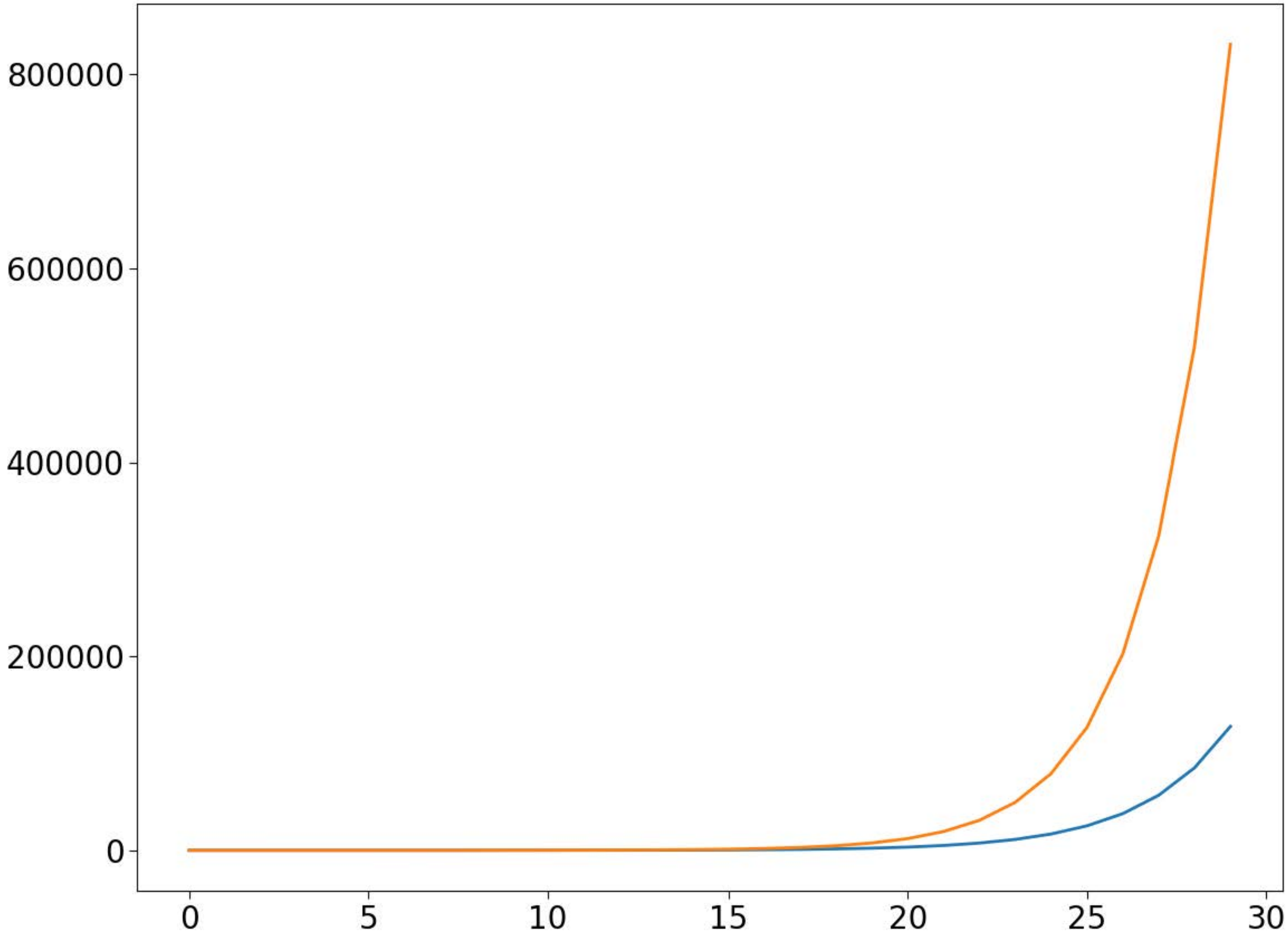
DISPLAY OF `cube`



DISPLAY OF `lin`



DISPLAY OF `expo`



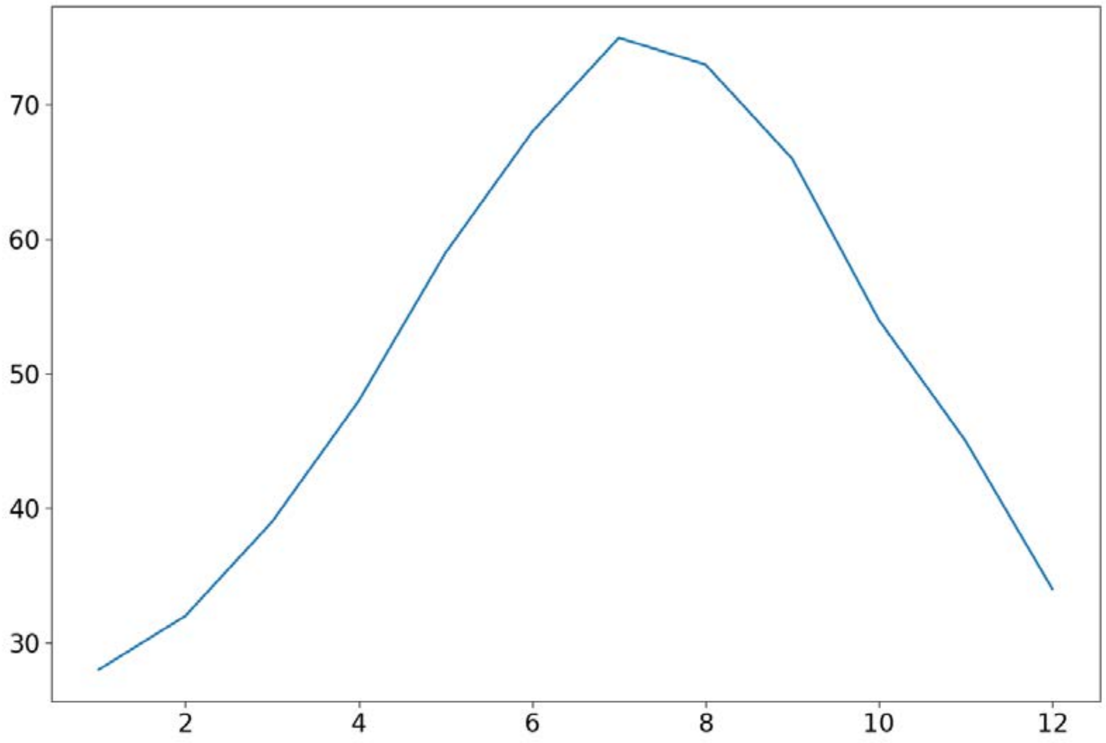
Second curve

Note how `matplotlib` automatically scales to fit both plots within frame

First curve

A "REAL" EXAMPLE

```
months = range(1, 13, 1)
temps = [28,32,39,48,59,68,75,73,66,54,45,34]
plt.plot(months, temps)
```



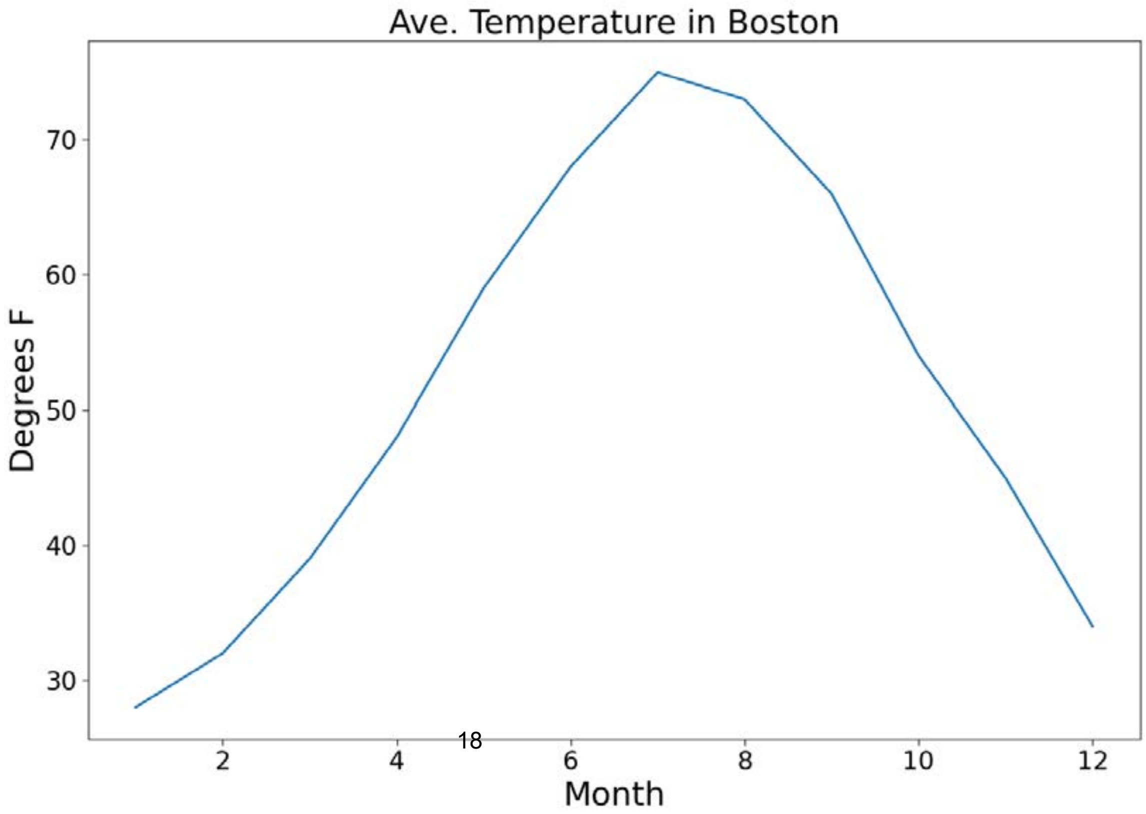
`matplotlib` has automatically selected x and y scales to best fit data

But what is this trying to tell us?
Suppose I just showed you the graph;
how do you know its meaning?

A "REAL" EXAMPLE

```
months = range(1, 13, 1)
temps = [28,32,39,48,59,68,75,73,66,54,45,34]
plt.plot(months, temps)
```

```
plt.title('Ave. Temperature in Boston')
plt.xlabel('Month')
plt.ylabel('Degrees F')
```



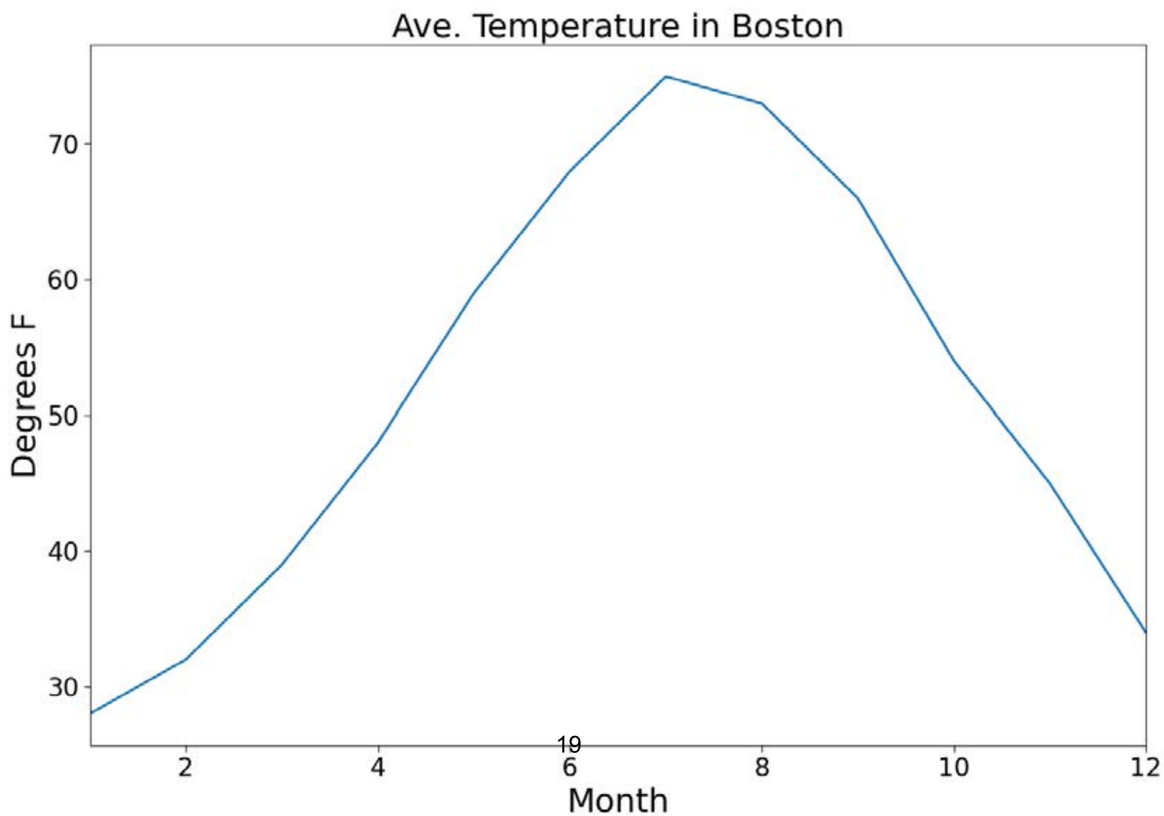
Still a bit weird looking

A "REAL" EXAMPLE

```
months = range(1, 13, 1)
temps = [28,32,39,48,59,68,75,73,66,54,45,34]
plt.plot(months, temps)

plt.title('Ave. Temperature in Boston')
plt.xlabel('Month')
plt.ylabel('Degrees F')
```

```
plt.xlim(1, 12)
```

This sets limits on display for x axis

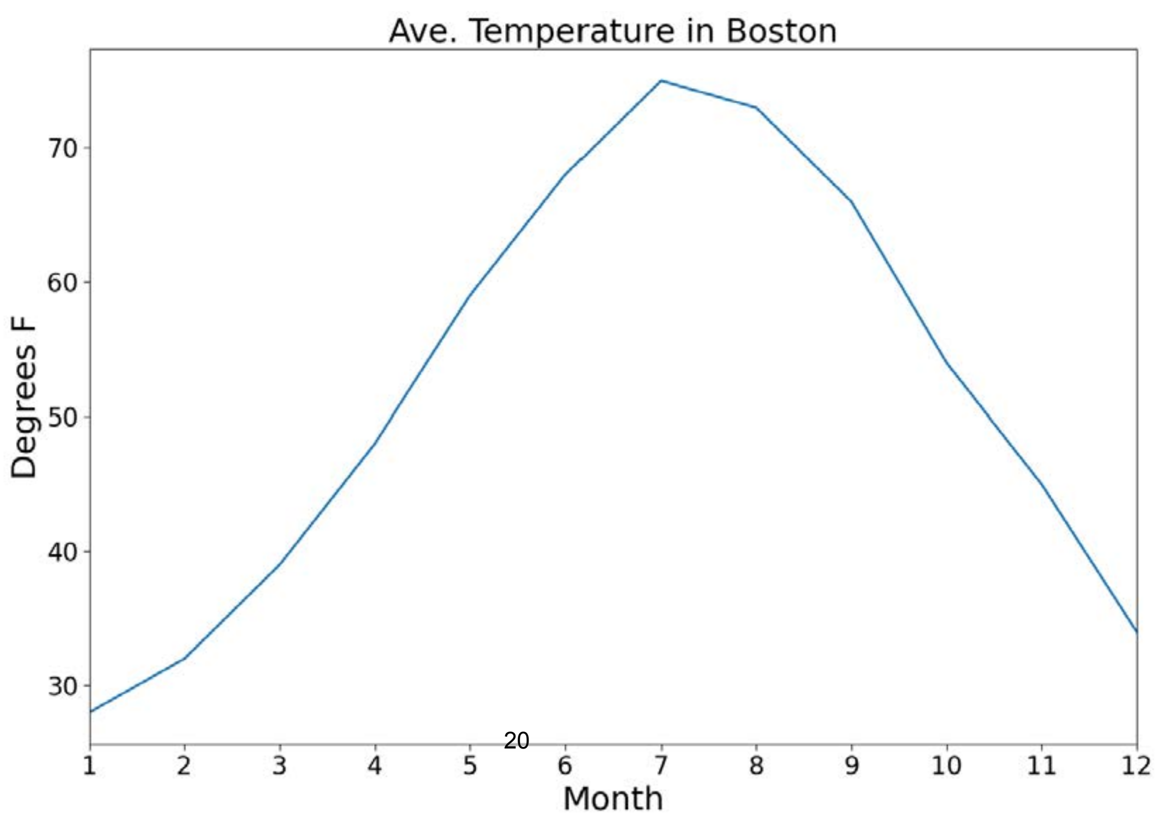


Suppose I want to see each month on x-axis?

A "REAL" EXAMPLE

```
months = range(1, 13, 1)
temps = [28,32,39,48,59,68,75,73,66,54,45,34]
plt.plot(months, temps)

plt.title('Ave. Temperature in Boston')
plt.xlabel('Month')
plt.ylabel('Degrees F')

plt.xticks((1,2,3,4,5,6,7,8,9,10,11,12))
```

This specifies which x values to mark



But what about those who can't map numbers to months?

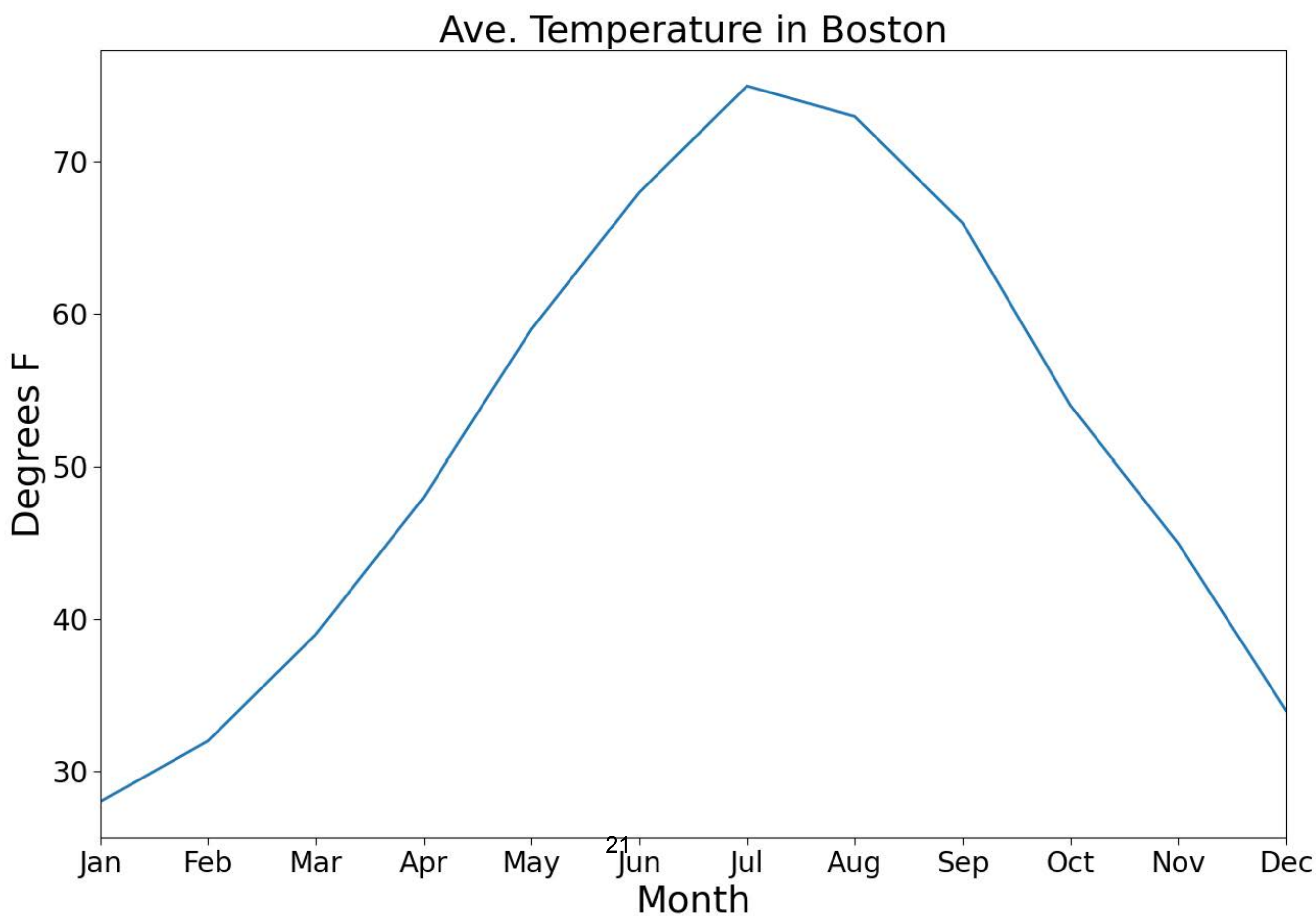
A "REAL" EXAMPLE

Locations of tick marks

The \ tells Python to continue the next line as part of same line

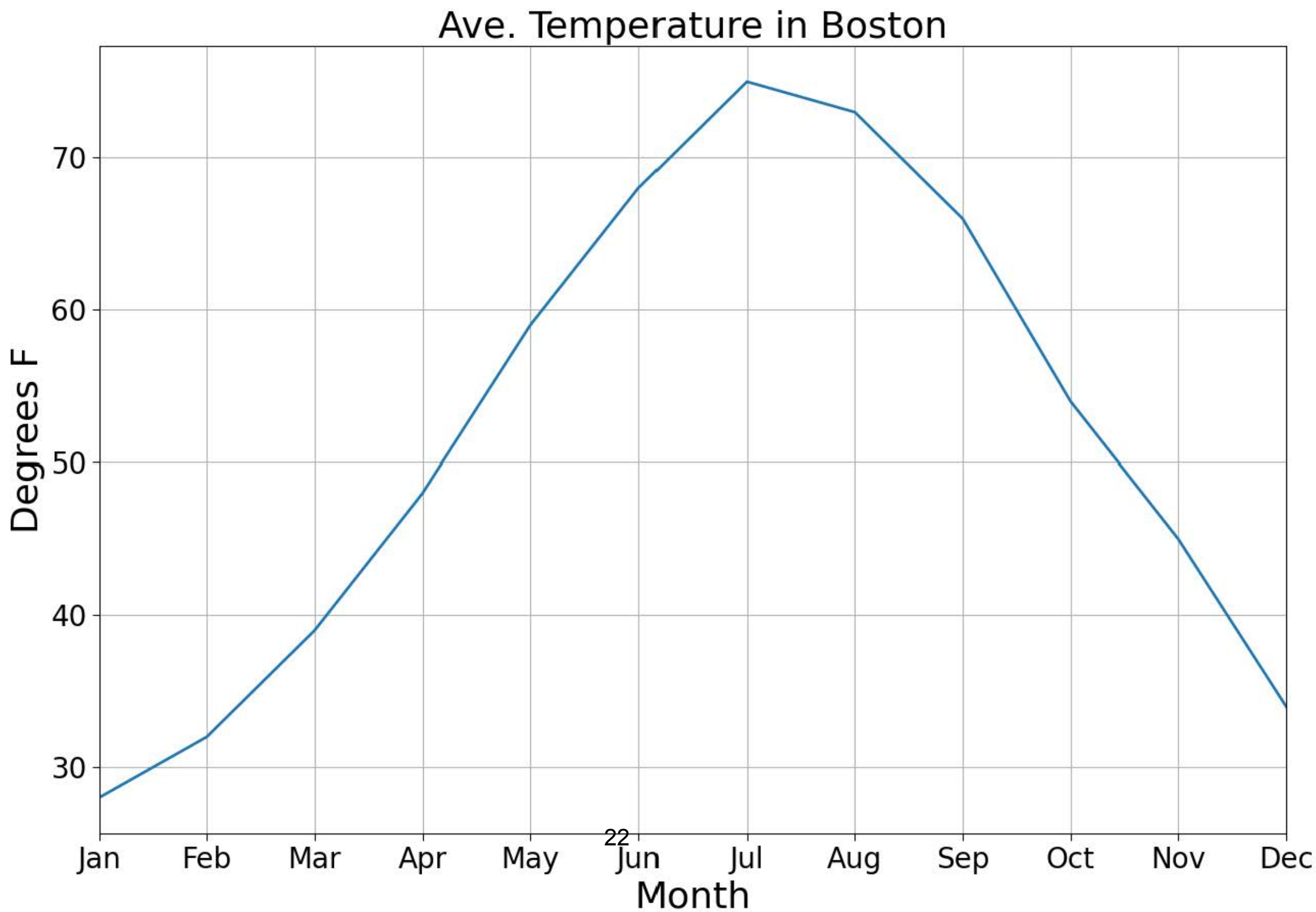
```
plt.xticks((1,2,3,4,5,6,7,8,9,10,11,12),
           ('Jan','Feb','Mar','Apr','May','Jun',\
            'Jul','Aug','Sep','Oct','Nov','Dec'))
```

Labels for tick marks



ADDING GRID LINES

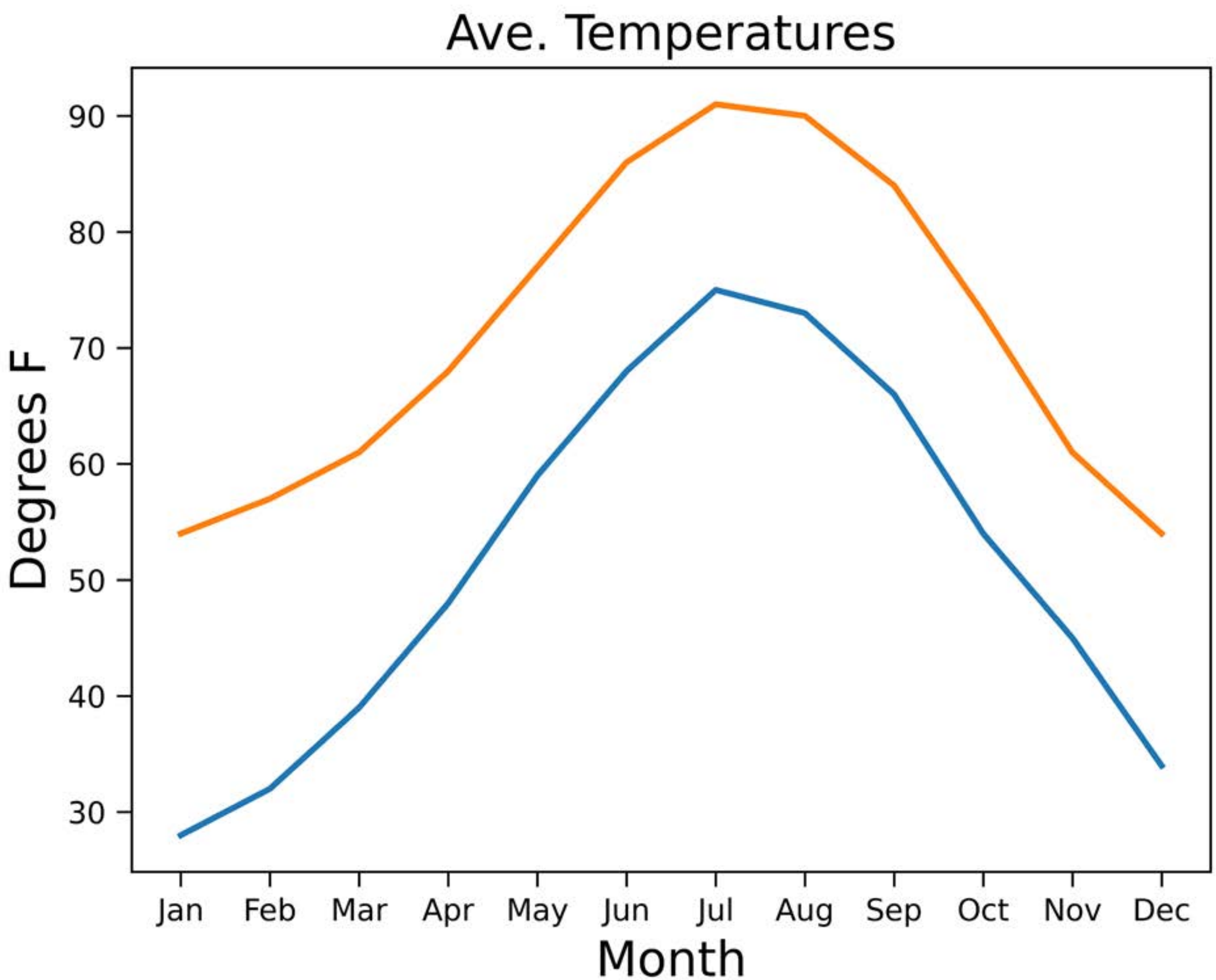
Can toggle grid lines on/off with `plt.grid()`



LET'S ADD ANOTHER CITY

```
months = range(1, 13, 1)
boston = [28,32,39,48,59,68,75,73,66,54,45,34]
plt.plot(months, boston )
phoenix = [54,57,61,68,77,86,91,90,84,73,61,54]
plt.plot(months, phoenix )
# Add labels and title
plt.title('Ave. Temperatures')
plt.xlabel('Month')
plt.ylabel('Degrees F')
```

BUT WHERE AM I?



LET'S ADD ANOTHER CITY

```
months = range(1, 13, 1)
boston = [28,32,39,48,59,68,75,73,66,54,45,34]
plt.plot(months, boston, label = 'Boston')
phoenix = [54,57,61,68,77,86,91,90,84,73,61,54]
plt.plot(months, phoenix, label = 'Phoenix')
# Add labels and title
plt.title('Ave. Temperatures')
plt.xlabel('Month')
plt.ylabel('Degrees F')
plt.legend(loc = 'best')
```

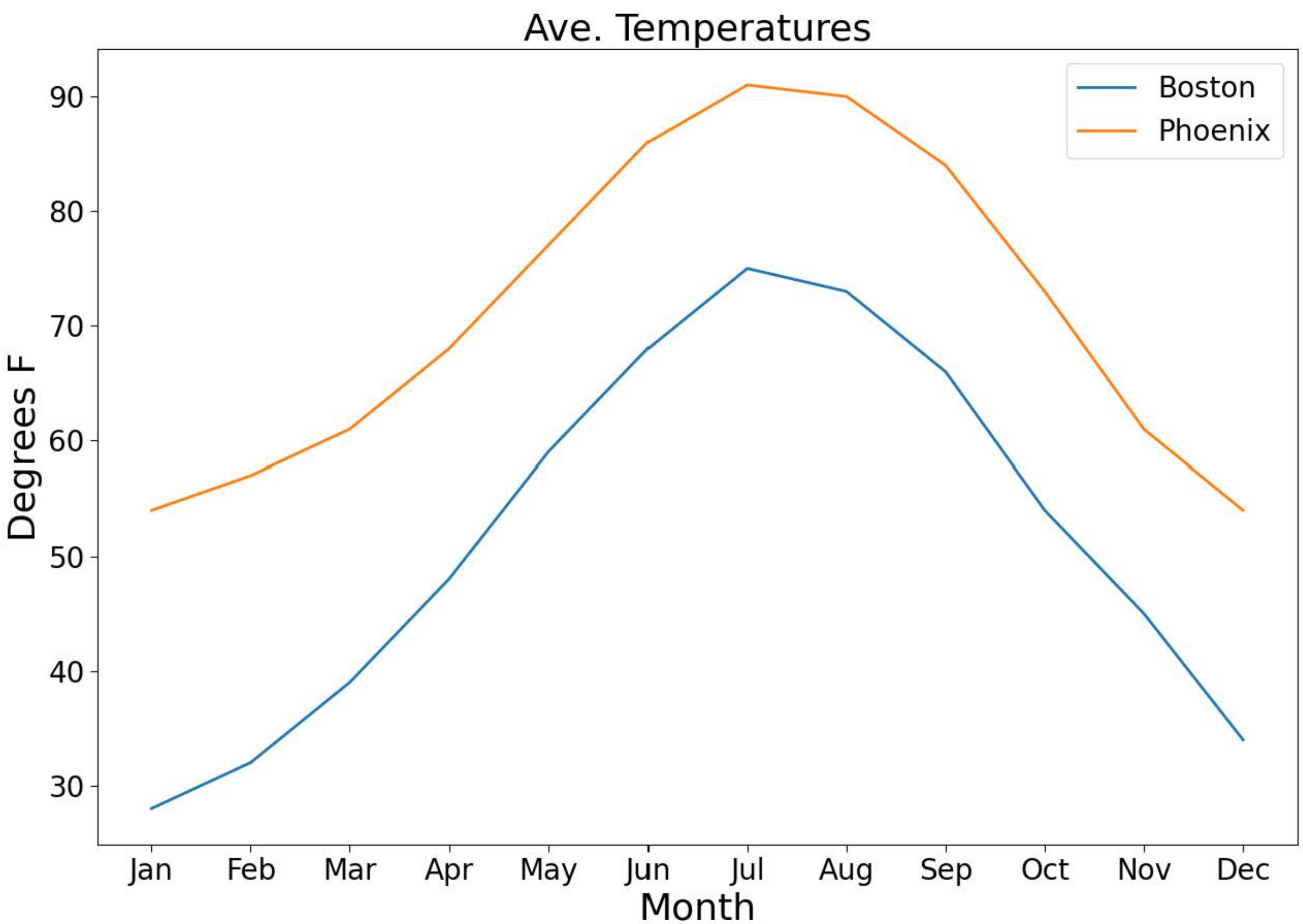
keyword

Choice for where to place legend

Other options:

- upper left
- upper right
- lower left
- lower right
- upper center
- lower center
- center right
- center left
- center

PLOT WITH TWO CURVES



Note: Python picked different colors for each plot; we could specify if we wanted

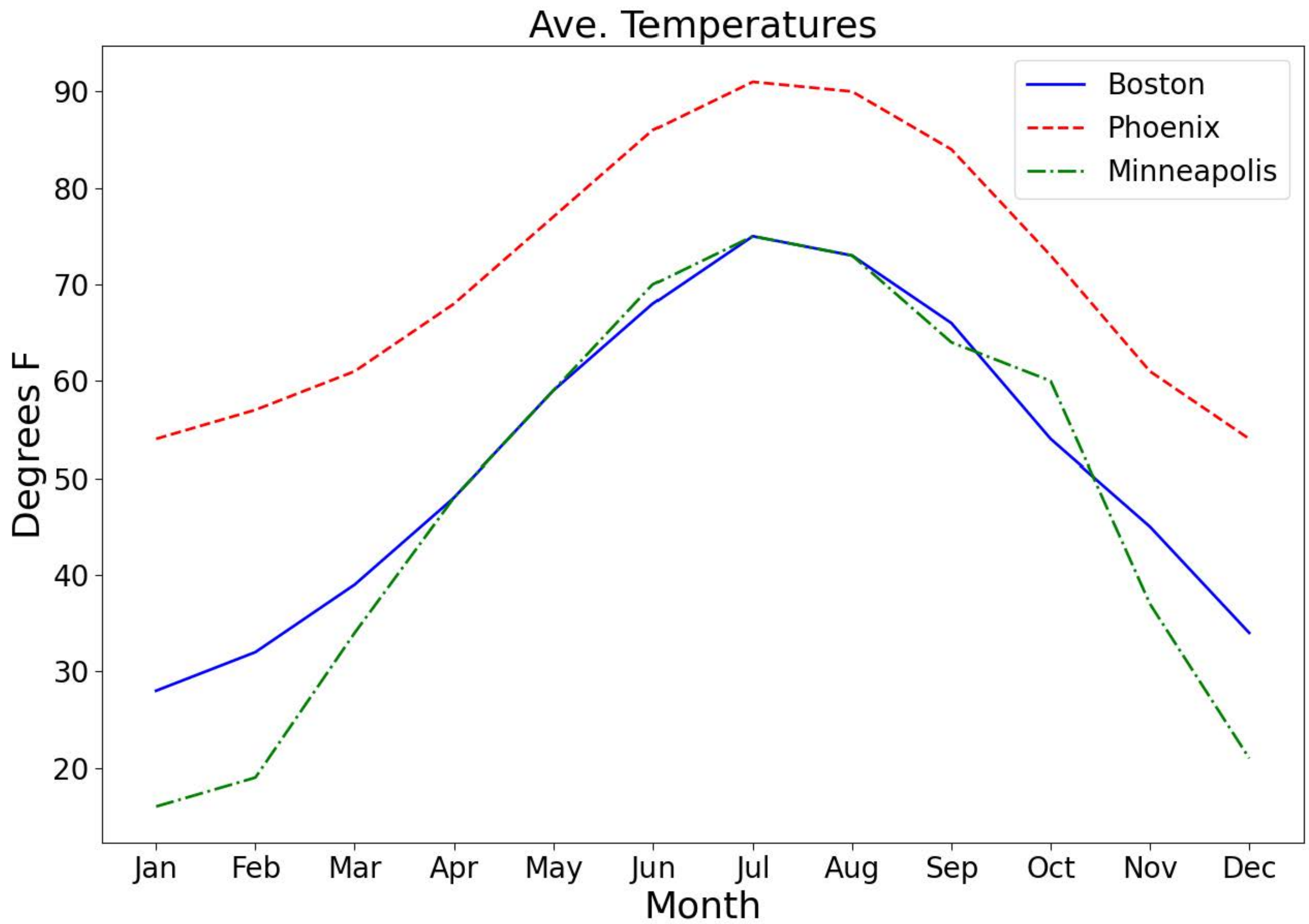
CONTROLLING PARAMETERS

- Suppose we want to control **details of the displays**
- Examples:
  - Changing **color** or style of data sets
  - Changing **width** of lines or displays
  - Using **subplots**
- Can provide a “format” argument to plot
  - “marker”, “line”, “color”
  - Can skip any of these choices, plot takes default
  - Order doesn’t matter, as no confusion between symbols

CONTROLLING COLOR AND STYLE

```
months = range(1, 13, 1)
boston = [28,32,39,48,59,68,75,73,66,54,45,34]
plt.plot(months, boston, 'b-', label = 'Boston')
phoenix = [54,57,61,68,77,86,91,90,84,73,61,54]
plt.plot(months, phoenix, 'r--', label = 'Phoenix')
msp = [16,19,34,48,59,70,75,73,64,60,37,21]
plt.plot(months, msp, 'g-.', label = 'Minneapolis')
plt.legend(loc = 'best', fontsize=20)
```

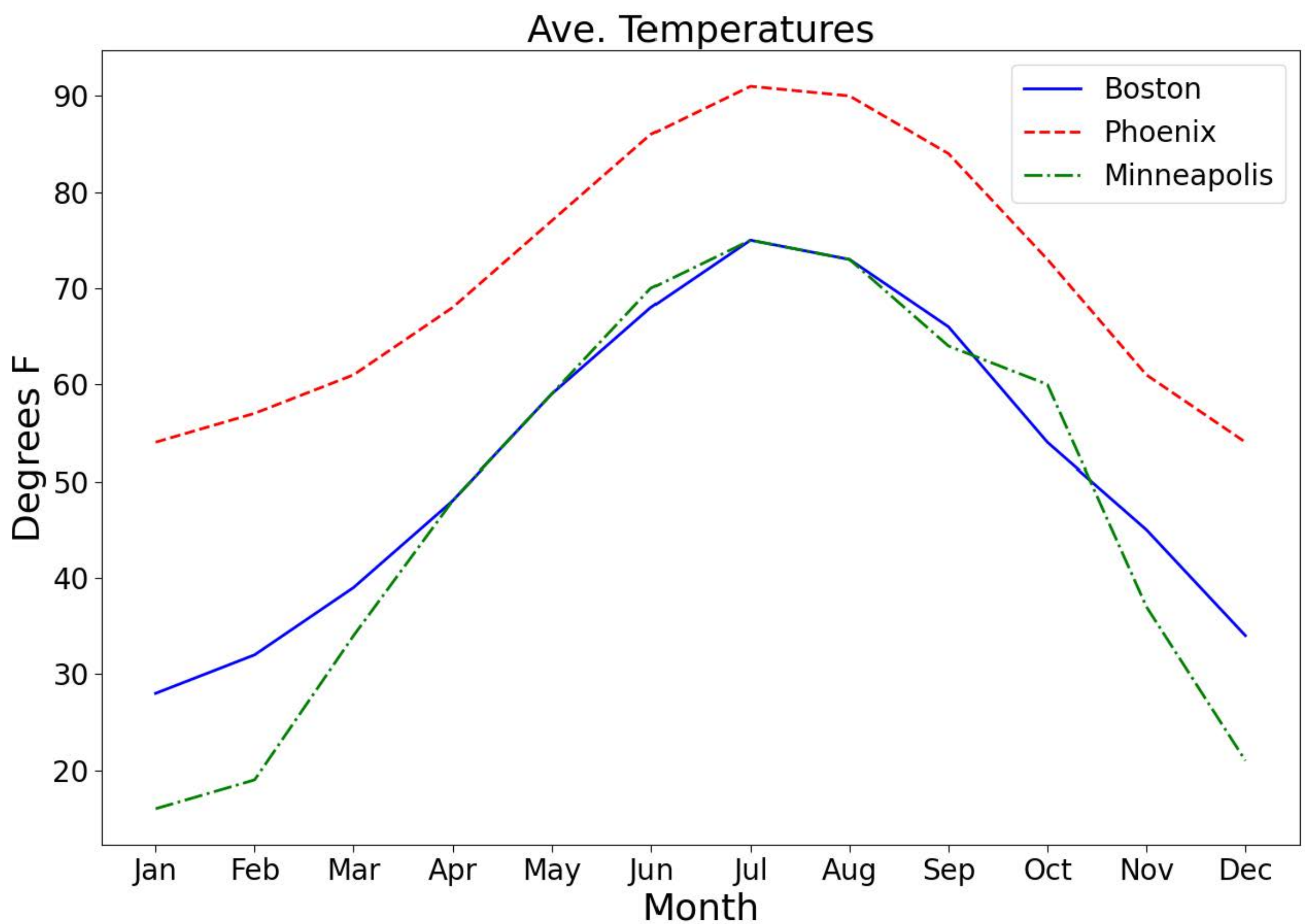
CONTROLLING COLOR AND STYLE



USING KEYWORDS

```
months = range(1, 13, 1)
boston = [28,32,39,48,59,68,75,73,66,54,45,34]
plt.plot(months, boston, label = 'Boston',\
            color = 'b', linestyle = '-')
phoenix = [54,57,61,68,77,86,91,90,84,73,61,54]
plt.plot(months, phoenix, label = 'Phoenix',\
            color = 'r', linestyle = '--')
msp = [16,19,34,48,59,70,75,73,64,60,37,21]
plt.plot(months, msp, label = 'Minneapolis',\
            color = 'g', linestyle = '-.')
plt.legend(loc = 'best', fontsize=20)
plt.title('Ave. Temperatures')
plt.xlabel('Month')
plt.ylabel(('Degrees F'))
plt.xticks((1,2,3,4,5,6,7,8,9,10,11,12),
            ('Jan','Feb','Mar','Apr','May','Jun',\
             'Jul','Aug','Sep','Oct','Nov','Dec'))
```

CONTROLLING COLOR AND STYLE



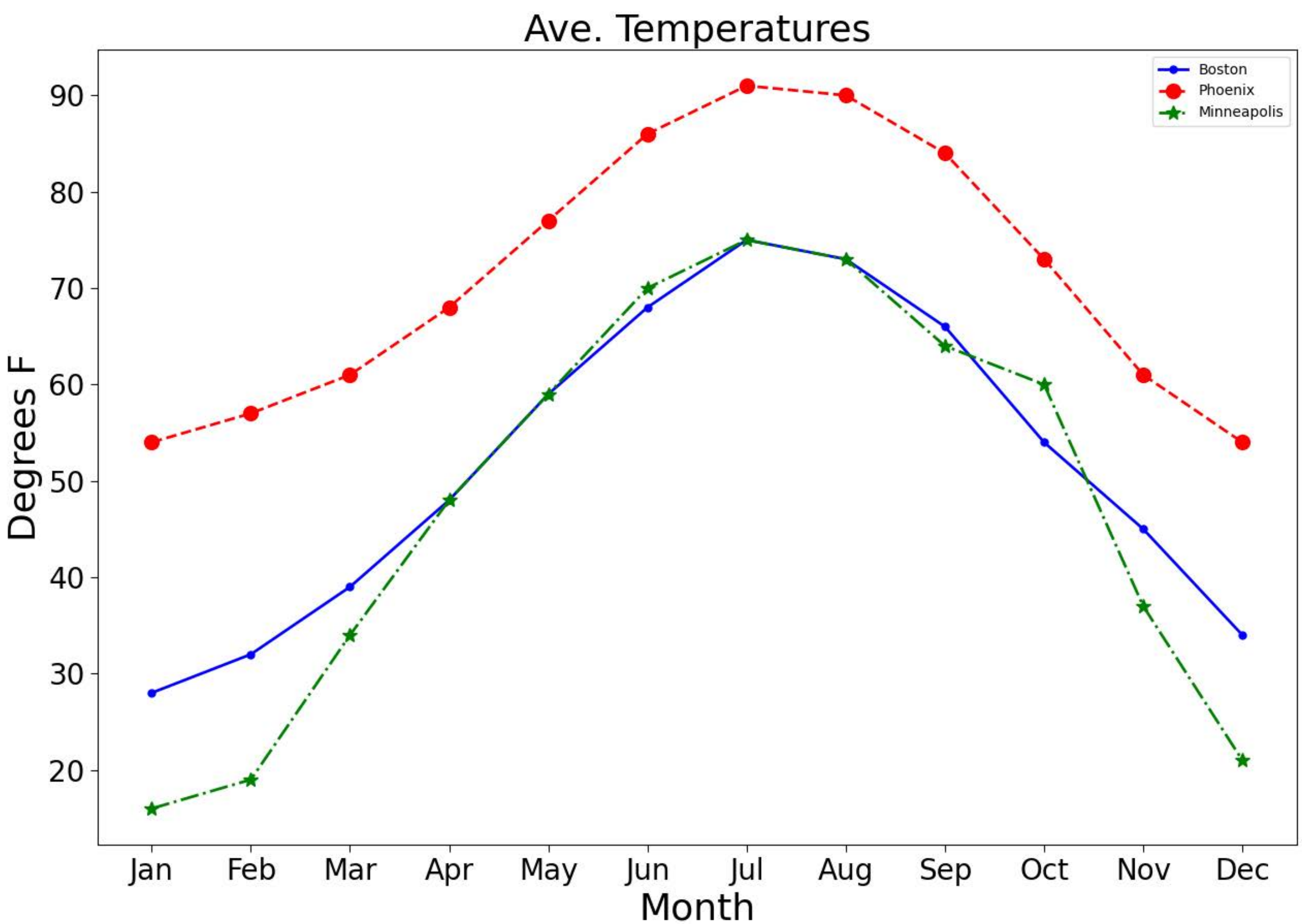
LINE, COLOR, MARKER OPTIONS

- Line Style
  - -	solid line
  - --	dashed line
  - -.	dash dot line
  - :	dotted line
- Color Options (plus many more)
  - b	blue
  - g	green
  - r	red
  - c	cyan
  - m	magenta
  - y	yellow
  - k	black
  - w	white
- Marker Options (plus many more)
  - .	point
  - o	circle
  - v	triangle down
  - ^	triangle up
  - *	star

CONTROLLING COLOR AND STYLE

```
months = range(1, 13, 1)
boston = [28,32,39,48,59,68,75,73,66,54,45,34]
plt.plot(months, boston, '.b-', label = 'Boston')
phoenix = [54,57,61,68,77,86,91,90,84,73,61,54]
plt.plot(months, phoenix, 'or--', label = 'Phoenix')
msp = [16,19,34,48,59,70,75,73,64,60,37,21]
plt.plot(months, msp, '*g-.', label = 'Minneapolis')
plt.legend(loc = 'best', fontsize=20)
```

WITH MARKERS



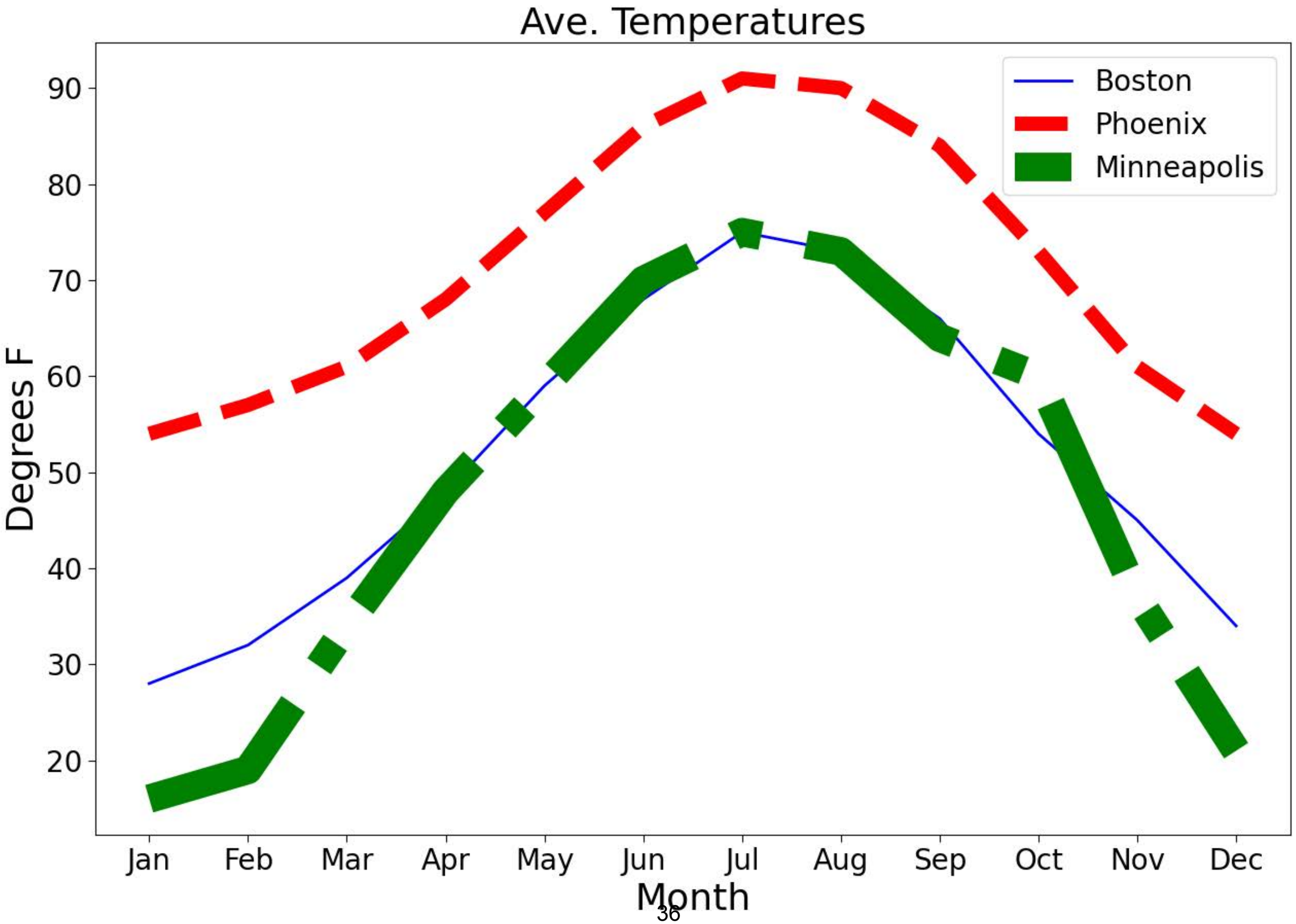
Note how actual points being plotted are now marked

CONTROLLING LINE WIDTH

```
months = range(1, 13, 1)
boston = [28,32,39,48,59,68,75,73,66,54,45,34]
plt.plot(months, boston, label = 'Boston',\
            color = 'b', linestyle = '-', linewidth = 2)
phoenix = [54,57,61,68,77,86,91,90,84,73,61,54]
plt.plot(months, phoenix, label = 'Phoenix',\
            color = 'r', linestyle = '--', linewidth = 10)
msp = [16,19,34,48,59,70,75,73,64,60,37,21]
plt.plot(months, msp, label = 'Minneapolis',\
            color = 'g', linestyle = '-.', linewidth = 20)
plt.legend(loc = 'best', fontsize=20)
```

MANY OTHER OPTIONS

- Using the linewidth keyword (in pixels)



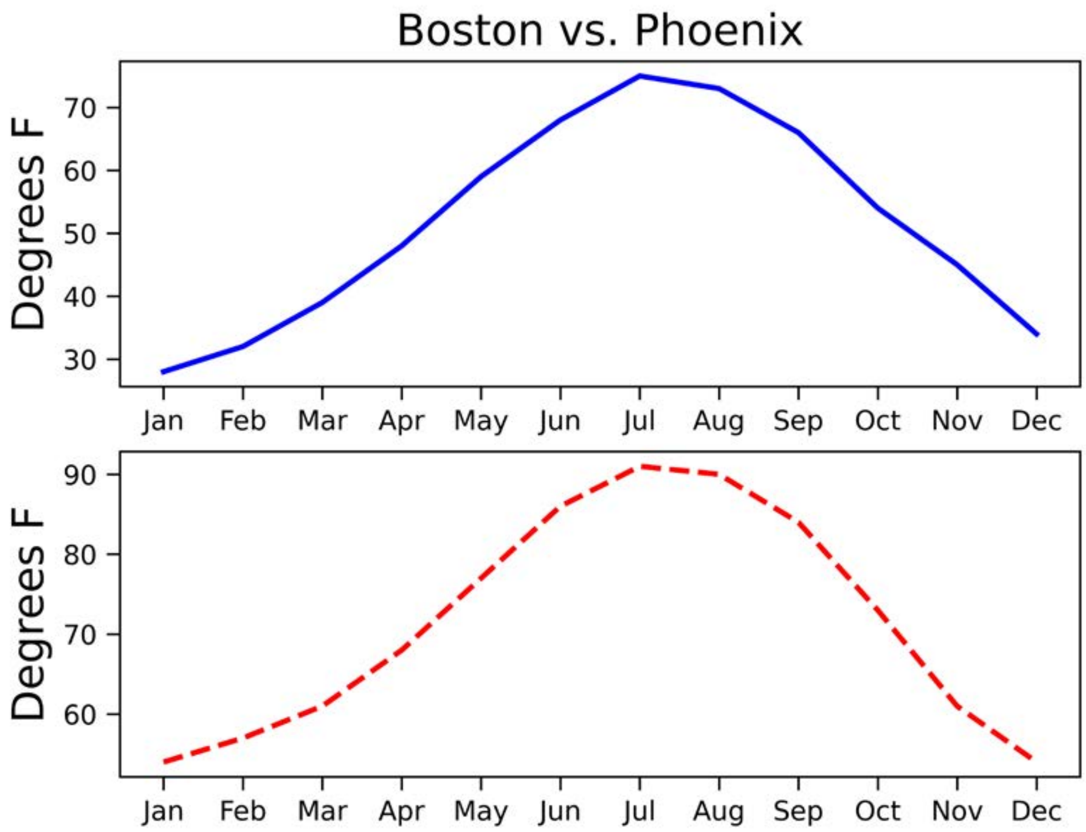
PLOTS WITHIN PLOTS

```
months = range(1, 13, 1)
boston = [28,32,39,48,59,68,75,73,66,54,45,34]
plt.subplot(2,1,1)
plt.plot(months, boston, 'b-')
plt.ylabel('Degrees F')
plt.title('Boston vs. Phoenix')
plt.xticks((1,2,3,4,5,6,7,8,9,10,11,12),
           ('Jan','Feb','Mar','Apr','May','Jun',\
            'Jul','Aug','Sep','Oct','Nov','Dec'))
phoenix = [54,57,61,68,77,86,91,90,84,73,61,54]
plt.subplot(2,1,2)
plt.plot(months, phoenix, 'r--')
plt.ylabel('Degrees F')
plt.xticks((1,2,3,4,5,6,7,8,9,10,11,12),
           ('Jan','Feb','Mar','Apr','May','Jun',\
            'Jul','Aug','Sep','Oct','Nov','Dec'))
```

Plot with 2 rows, 1 column, this is first

Plot with 2 rows, 1 column, this is second

AND THE PLOT THICKENS



But this can be misleading?

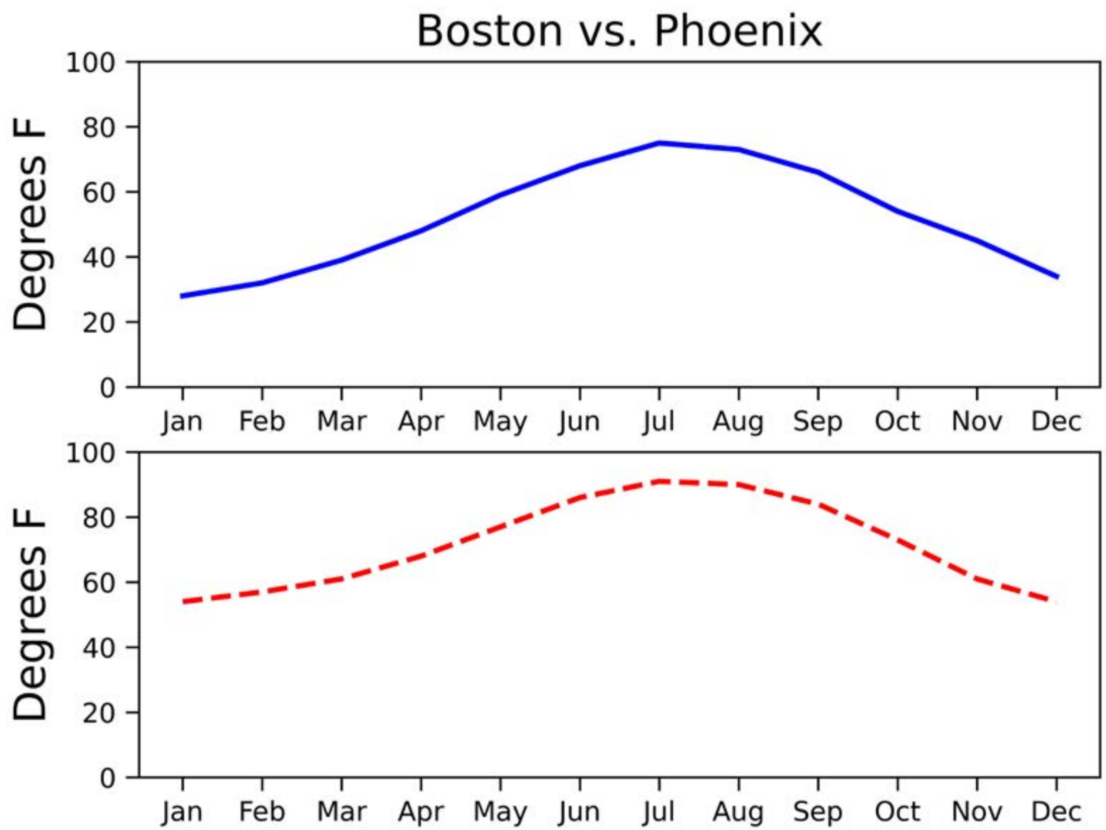
Y scales are different!

PLOTS WITHIN PLOTS

Fix y axis so plots are similar

```
months = range(1, 13, 1)
boston = [28,32,39,48,59,68,75,73,66,54,45,34]
plt.subplot(2,1,1)
plt.ylim(0, 100)
plt.plot(months, boston, 'b-')
plt.ylabel('Degrees F')
plt.title('Boston vs. Phoenix')
plt.xticks((1,2,3,4,5,6,7,8,9,10,11,12),
           ('Jan','Feb','Mar','Apr','May','Jun',\
            'Jul','Aug','Sep','Oct','Nov','Dec'))
phoenix = [54,57,61,68,77,86,91,90,84,73,61,54]
plt.subplot(2,1,2)
plt.ylim(0, 100)
plt.plot(months, phoenix, 'r--')
plt.ylabel('Degrees F')
plt.xticks((1,2,3,4,5,6,7,8,9,10,11,12),
           ('Jan','Feb','Mar','Apr','May','Jun',\
            'Jul','Aug','Sep','Oct','Nov','Dec'))
```

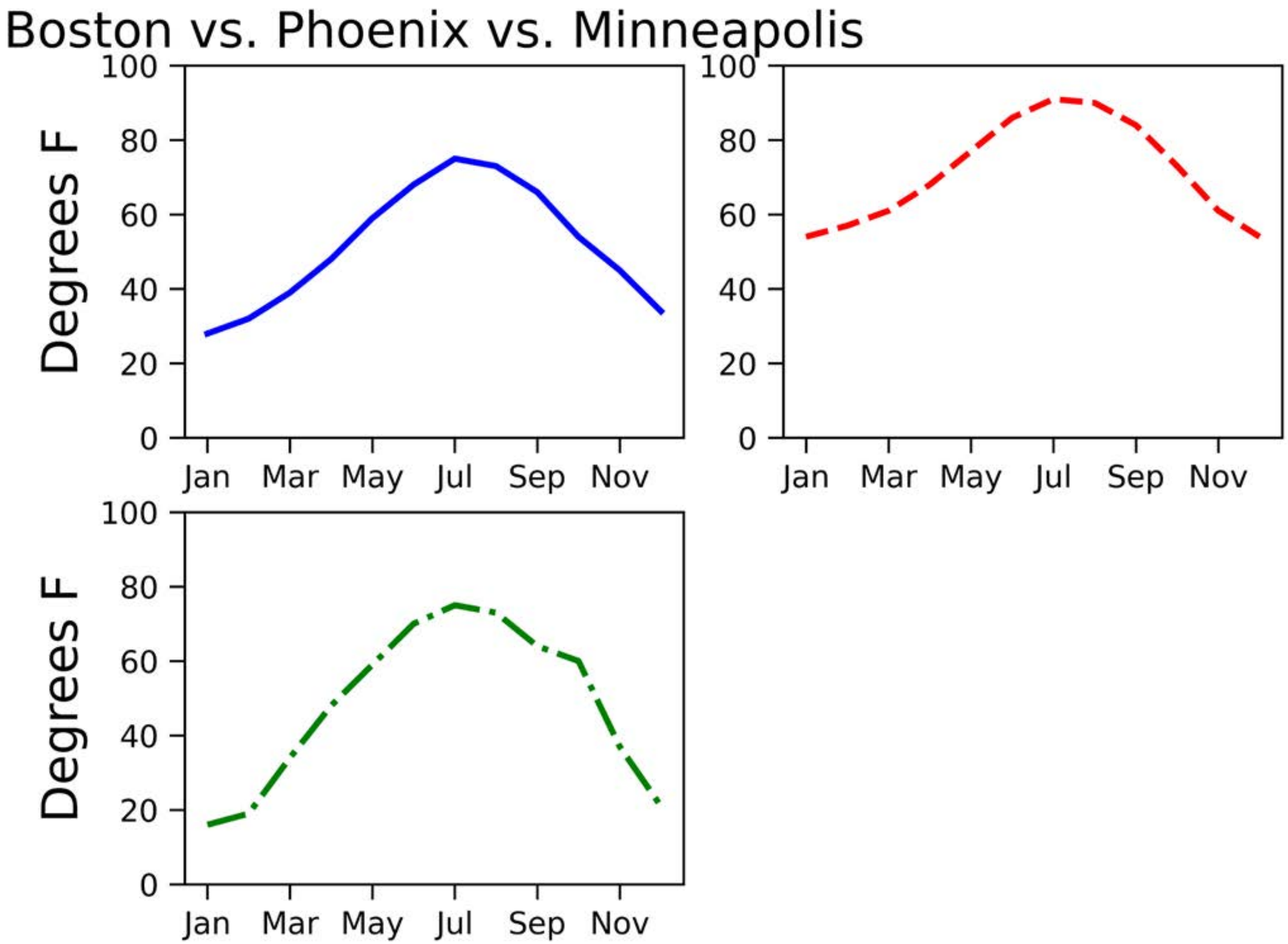
AND THE PLOT THICKENS



LOTS OF SUBPLOTS

```
boston = [28,32,39,48,59,68,75,73,66,54,45,34]
plt.subplot(2,2,1)
plt.ylim(0, 100)
plt.plot(months, boston, 'b-')
plt.ylabel('Degrees F')
plt.title('Boston')
plt.xticks((1,3,5,7,9,11),('Jan','Mar','May','Jul','Sep','Nov'))

phoenix = [54,57,61,68,77,86,91,90,84,73,61,54]
plt.subplot(2,2,2)
plt.ylim(0, 100)
plt.plot(months, phoenix, 'r--')
plt.title('Phoenix')
plt.xticks((1,3,5,7,9,11),('Jan','Mar','May','Jul','Sep','Nov'))

msp = [16,19,34,48,59,70,75,73,64,60,37,21]
plt.subplot(2,2,3)
plt.ylim(0, 100)
plt.plot(months, msp, 'g-.')
plt.ylabel('Degrees F')
plt.title('Minneapolis')
plt.xticks((1,3,5,7,9,11),('Jan','Mar','May','Jul','Sep','Nov'))
```

AND THE PLOT THICKENS



US POPULATION EXAMPLE

A MORE INTERESTING EXAMPLE

- Let's try plotting some more complicated data
- We have provided a file with the US population recorded every 10 years for four centuries
- Would like to use plotting to examine that data
  - Use plotting to help **visualize** trends in the data
  - Use plotting to raise questions that might be **tested computationally** (you'll see much more of this if you take 6.100B)

THE INPUT FILE
USPopulation.txt

```
1610 350
1620 2,302
1630 4,646
1640 26,634
1650 50,368
1660 75,058
1670 111,935
1680 151,507
1690 210,372
1700 250,888
1710 331,711
1720 466,185
1730 629,445
1740 905,563
...

1960 179,323,175
1970 203,211,926
1980 226,545,805
1990 248,709,873
2000 281,421,906
2010 308,745,538
```

PLOTTING THE DATA

```
1610 350
1620 2,302
1630 4,646
1640 26,634
```

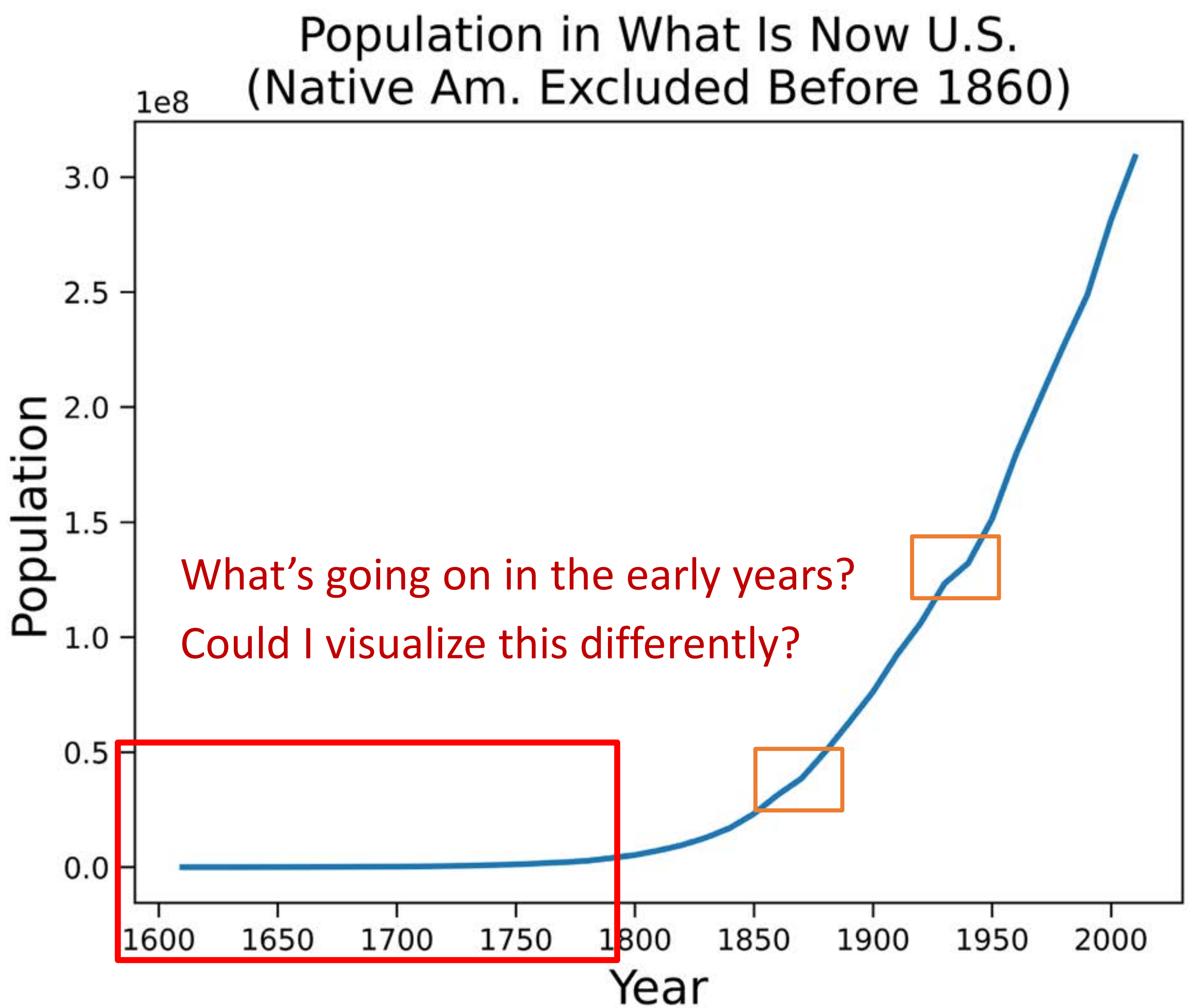
```
def getUSPop(fileName):
    inFile = open(fileName, 'r')
    dates, pops = [], []
    for l in inFile:
        line = ''
        for c in l:
            if c in '0123456789 ':
                line += c
        line = line.split(' ')
        dates.append(int(line[0]))
        pops.append(int(line[1]))
    return dates, pops

dates, pops = getUSPop('lec25_USPopulation.txt')
plt.plot(dates, pops)
plt.title('Population in What Is Now U.S.\n' +\
          '(Native Am. Excluded Before 1860)')
plt.xlabel('Year')
plt.ylabel('Population')
```

Remove commas for each line

Split into date and population

Convert to ints, and add to lists

POPULATION GROWTH



Visualizing data can expose things not easily seen in raw data

Impact of WWII

Impact of Civil War

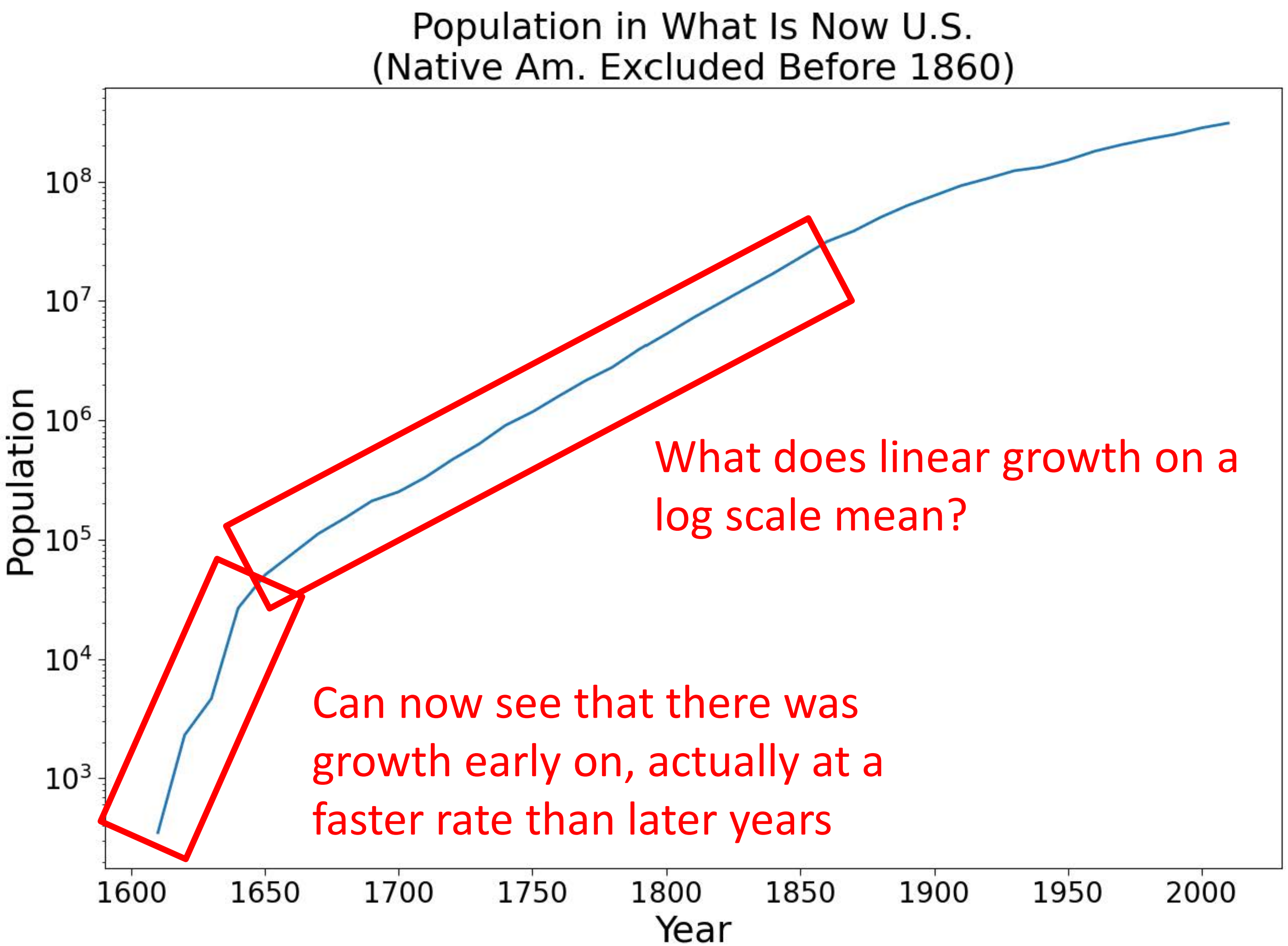
CHANGING THE SCALING

```
dates, pops = getUSPop('USPopulation.txt')
plt.plot(dates, pops)
plt.title('Population in What Is Now U.S\n' +\
          '(Native Am. Excluded Before 1860)')
plt.xlabel('Year')
plt.ylabel('Population')
plt.semilogy()
```

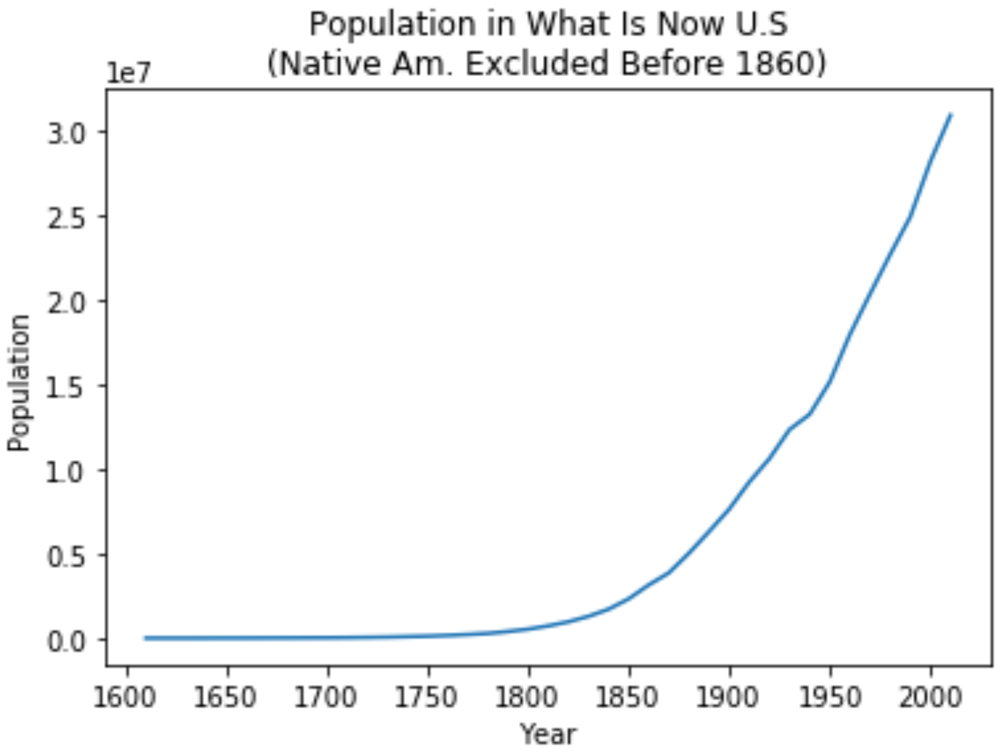
Use log scale on y axis

Log scale means each increment along axis corresponds to exponential increase in size; while in normal scale each increment corresponds to linear increase in size

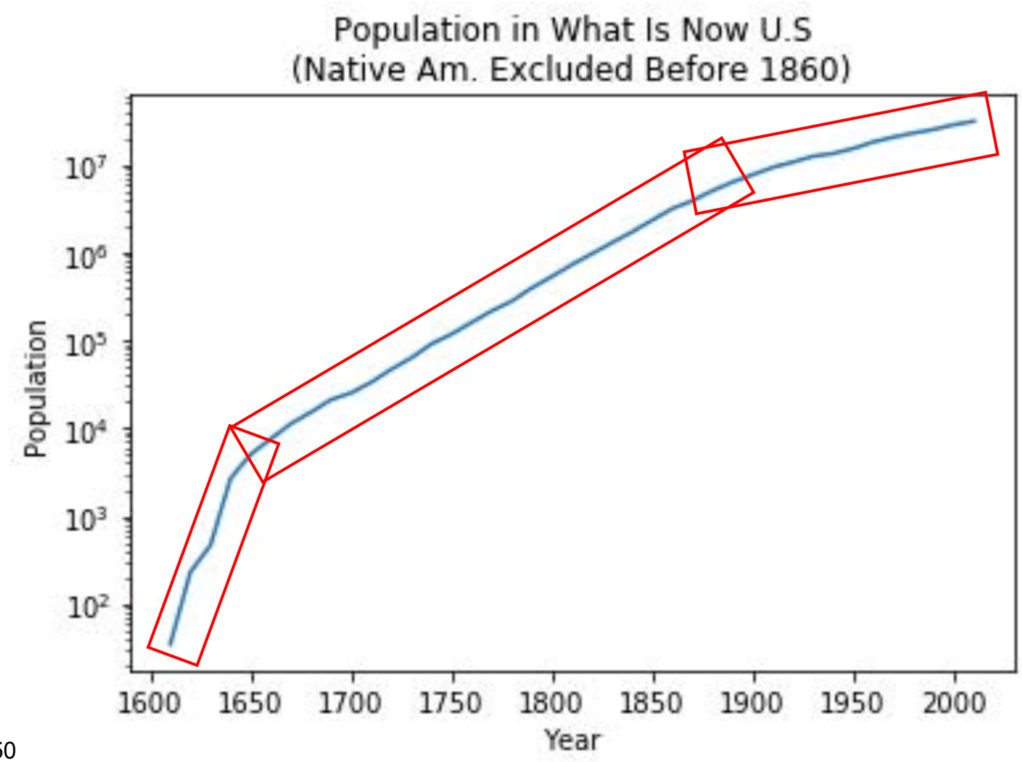
POPULATION GROWTH



WHICH DO YOU FIND MORE INFORMATIVE?



Changing visualization can help expose trends in data not seen with standard plotting



Visualization can raise questions: for ex. by eye, it appears that there are three different exponential growth periods

COUNTRY POPULATION EXAMPLE

THE DATA FILE

`countryPops.txt`

Notice pesky commas again in this value

```
1    China    1,379,302,771    July 2017 est.
2    India    1,281,935,911    July 2017 est.
3    United States   326,625,791 July 2017 est.
4    Indonesia   260,580,739 July 2017 est.
5    Brazil  207,353,391 July 2017 est.
6    Pakistan    204,924,861 July 2017 est.
7    Nigeria 190,632,261 July 2017 est.
8    Bangladesh  157,826,578 July 2017 est.
9    Russia  142,257,519 July 2017 est.
10   Japan   126,451,398 July 2017 est.

 ...
228 Montserrat  5,292   July 2017 est.
229 Falkland Islands (Islas Malvinas)   2,931   2014 est.
230 Svalbard    2,667   July 2016 est.
231 Norfolk Island  2,210   July 2014 est.
232 Christmas Island    2,205   July 2016 est.
233 Niue    1,626   June 2015 est.
234 Tokelau 1,285   2016 est.
235 Holy See (Vatican City) 1,000   2015 est.
236 Cocos (Keeling) Islands 596 July 2014 est.
237 Pitcairn Islands    54  July 2016 est.
```

Interested in analyzing the population numbers. Don't care about rank, country, or year.

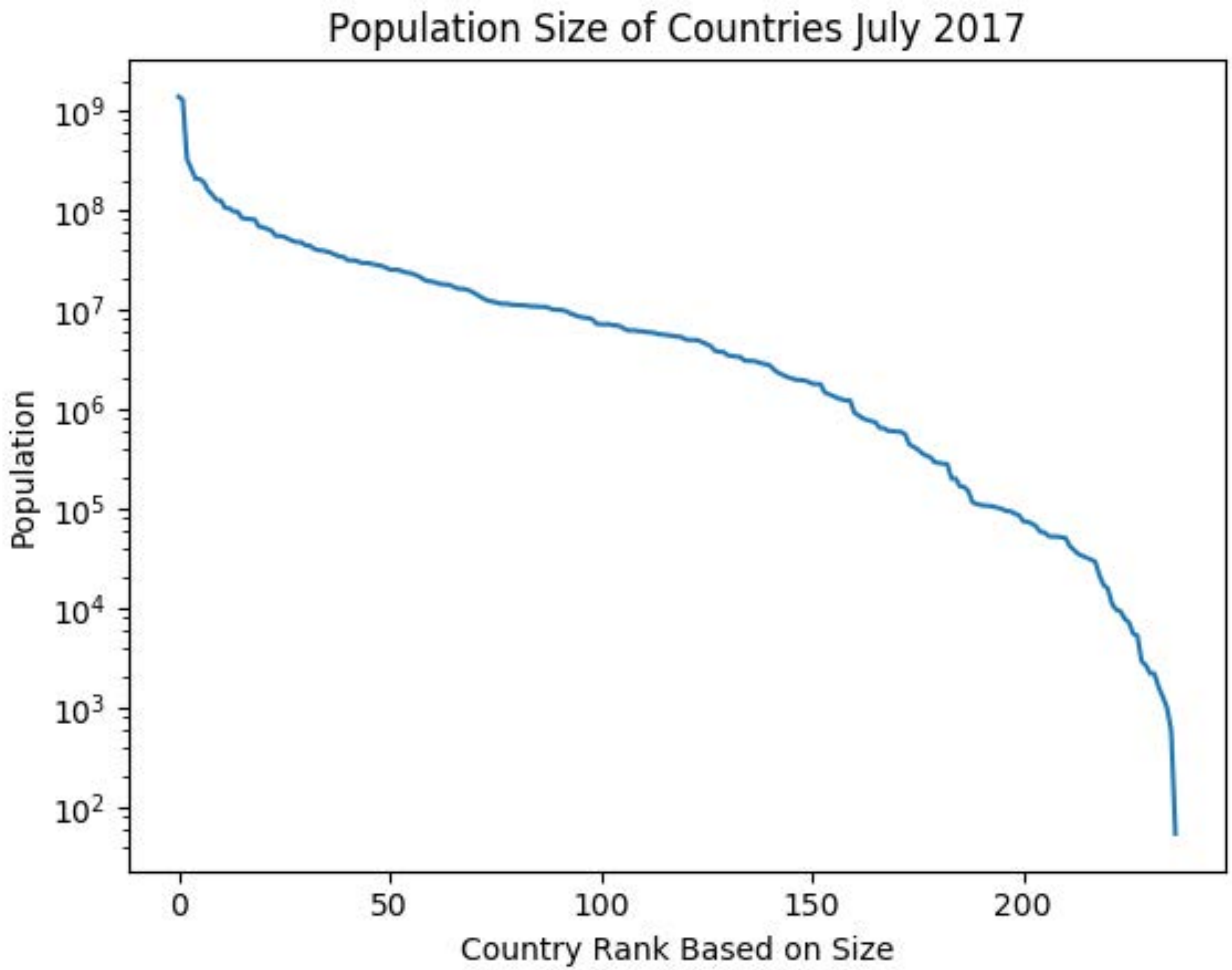
LOADING AND PLOTTING THE DATA

```
1    China    1,379,302,771    July 2017 est.
2    India    1,281,935,911    July 2017 est.
3    United States    326,625,791 July 2017 est.
4    Indonesia    260,580,739 July 2017 est.
```

```
def getCountryPops(fileName):
    inFile = open(fileName, 'r')
    pops = []
    for l in inFile:
        line = l.split('\t')
        l = line[2]
        pop = ''
        for c in l:
            if c in '0123456789':
                pop += c
        pops.append(int(pop))
    return pops

pops = getCountryPops('lec25_countryPops.txt')

plt.plot(pops)
plt.title('Population Size of Countries July 2017')
plt.ylabel('Population')
plt.xlabel('Country Rank Based on Size')
plt.semilogy()
```

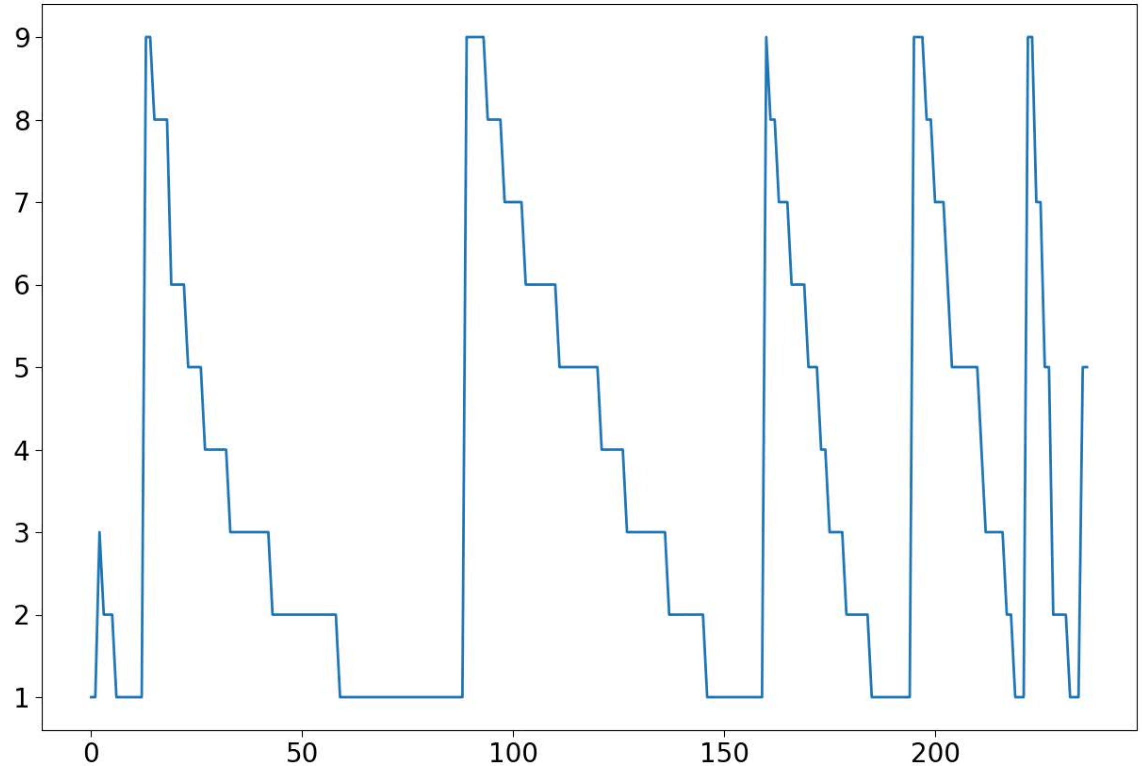
Grab only the population number column

POPULATION SIZES



STRANGE INVESTIGATION: FIRST DIGITS

```
pops = getCountryPops('lec25_countryPops.txt')
firstDigits = []
for p in pops:
    firstDigits.append(int(str(p)[0]))
### Plot the fist digits, as found in order in the file
plt.plot(firstDigits)
```



Why the saw tooth pattern?

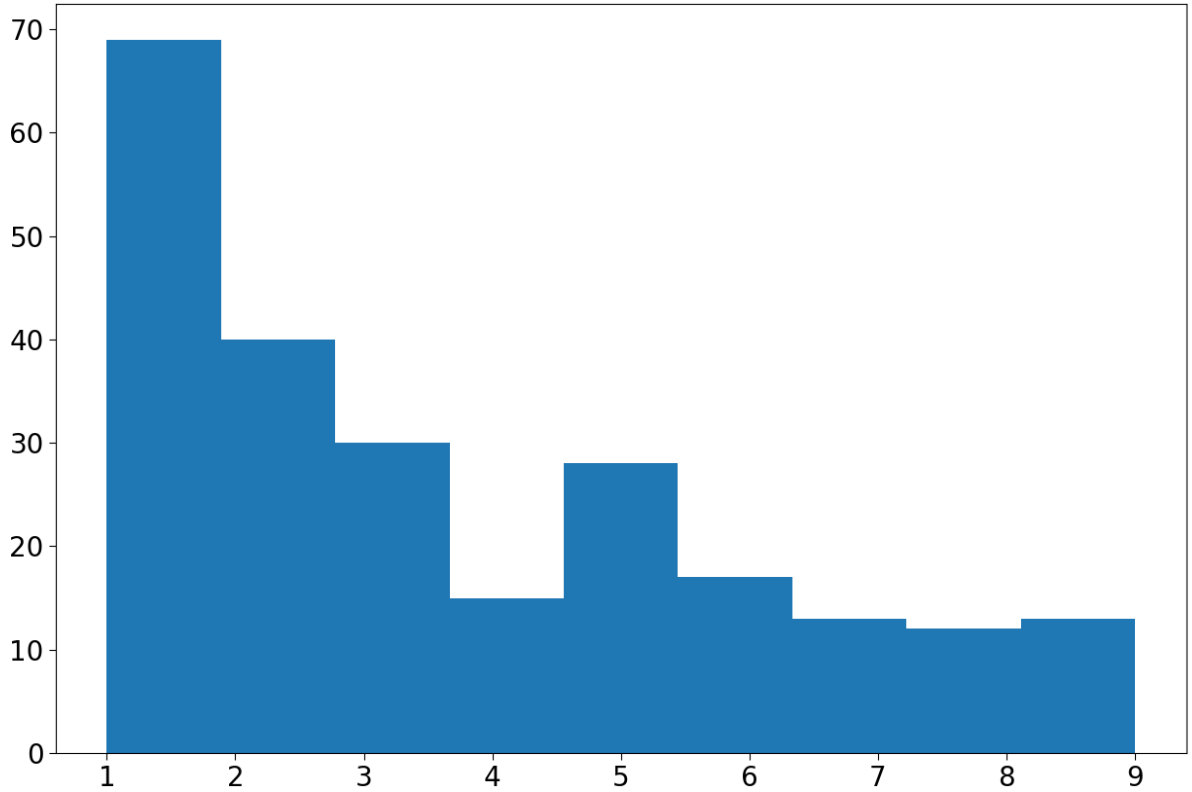
Countries are in order of biggest pop to smallest pop.

First digit pattern is: 9,8,7,...,2,1,9,8,7,6,5,...

FREQUENCY OF EACH DIGIT

```
plt.hist(firstDigits, bins = 9)
```

Surprising?
28% 1's



Benford's Law

$$P(d) = log_{10}(1+\frac{1}{d})$$

Many datasets follow this:

- # social media followers
- Stock values
- Grocery prices
- Sports stats
- Building heights
- Taxes paid

COMPARING CITIES EXAMPLE

AN EXTENDED EXAMPLE

- Let's use another example to examine how plotting allows us to explore data in different ways, and how it provides a valuable way to visualize that data
- Won't be looking at the code in detail
- Example data set
  - Mean daily temperature for each day for 55 years for 21 different US cities
  - Want to explore variations across years, and across cities

THE DATA FILE

temperatures.csv

```
CITY,TEMP,DATE
SEATTLE,3.1,19610101
SEATTLE,0.55,19610102
SEATTLE,0,19610103
SEATTLE,4.45,19610104
SEATTLE,8.35,19610105
SEATTLE,6.7,19610106
SEATTLE,9.7,19610107
SEATTLE,7.2,19610108
SEATTLE,9.45,19610109

...

CHICAGO,9.7,20151223
CHICAGO,3.35,20151224
CHICAGO,3.35,20151225
CHICAGO,4.2,20151226
CHICAGO,3.05,20151227
CHICAGO,1.7,20151228
CHICAGO,1.15,20151229
CHICAGO,-2.15,20151230
CHICAGO,-3.8,20151231
```

Temp in Celsius

Date in YYYYMMDD

temperatures.csv

CITY,TEMP,DATE
SEATTLE,3.1,19610101
SEATTLE,0.55,19610102
SEATTLE,0,19610103
SEATTLE,4.45,19610104

EXTRACTING DATA

This will return a list of temperatures (in F) and a corresponding list of dates for a specific city

```
def CtoF(c):
    return (c * 9/5) + 32

def getTempsForCity(city):
    inFile = open('temperatures.csv')
    temps = []
    dates = []
    for l in inFile:
        data = l.split(',')
        c = data[0]
        tem = data[1]
        date = data[2]
        if c == city:
            temps.append(CtoF(float(tem)))
            dates.append(date)
    return temps, dates
```

Only want temp for a specific city

File stores data as str, need to convert

AVERAGE TEMPERATURES

This will calculate the average temp over every day for 55 years, for every city.

```
def getAverageTemps():
    cities = getCities()[1:]
    xPts = range(len(cities))
    aveTemp = []
    cityLabels = []
    for c in cities:
        temps, dates = getTempsForCity(c)
        aveTemp.append(sum(temps)/len(temps))
        cityLabels.append(c[0:2])
        print(c[0:2], sum(temps)/len(temps))

    plt.figure('Temps')
    plt.scatter(xPts, aveTemp)
    plt.title('Ave. Temperatures')
    plt.xlabel('City')
    plt.ylabel(('Degrees F'))
    plt.xticks(xPts, cityLabels)
```

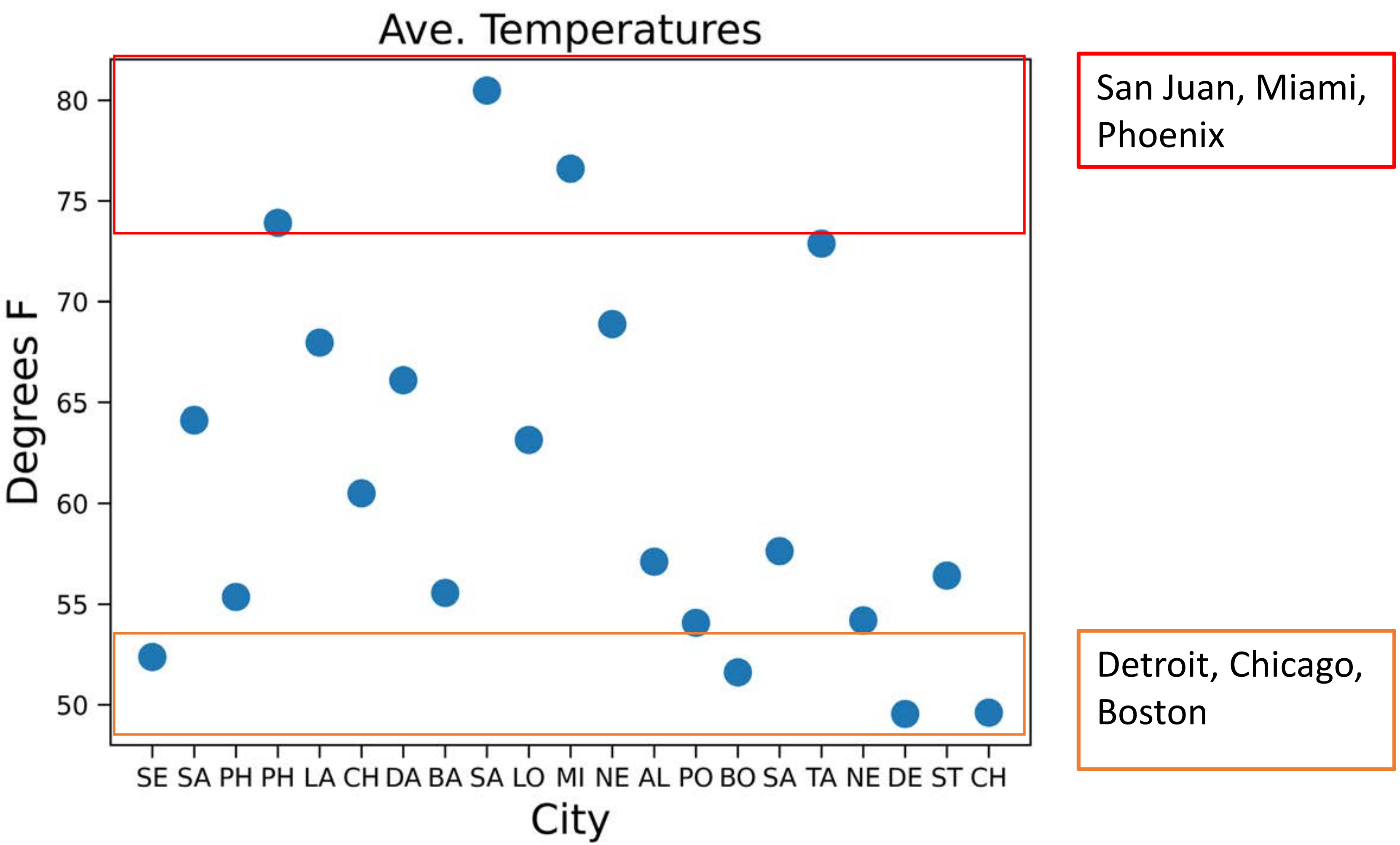
Get list of cities

Compute average temperature

Using first two characters as label

Just plotting points as a scatter plot (no connecting lines)

AND THE TEMPERATURE IS …



BUT MORE INTERESTING TO LOOK AT CHANGE OVER TIME

For one city, calculate the average temperature over each year.

```
def getTempsForYear(tem, dat, y):
    yearlyTemps = []
    for i in range(len(tem)):
        if y == dat[i][:4]:
            yearlyTemps.append(tem[i])
    return sum(yearlyTemps)/len(yearlyTemps), y
```

Check that entry is for right year

```
def getTempsByYearForCity(city):
    temps, dates = getTempsForCity(city)
    averages = []
    years = []
    for y in range(1961,2016):
        tem = getTempsForYear(temps, dates, str(y))[0]
        averages.append(tem)
        years.append(str(y))
    return averages, years
```

Previous code

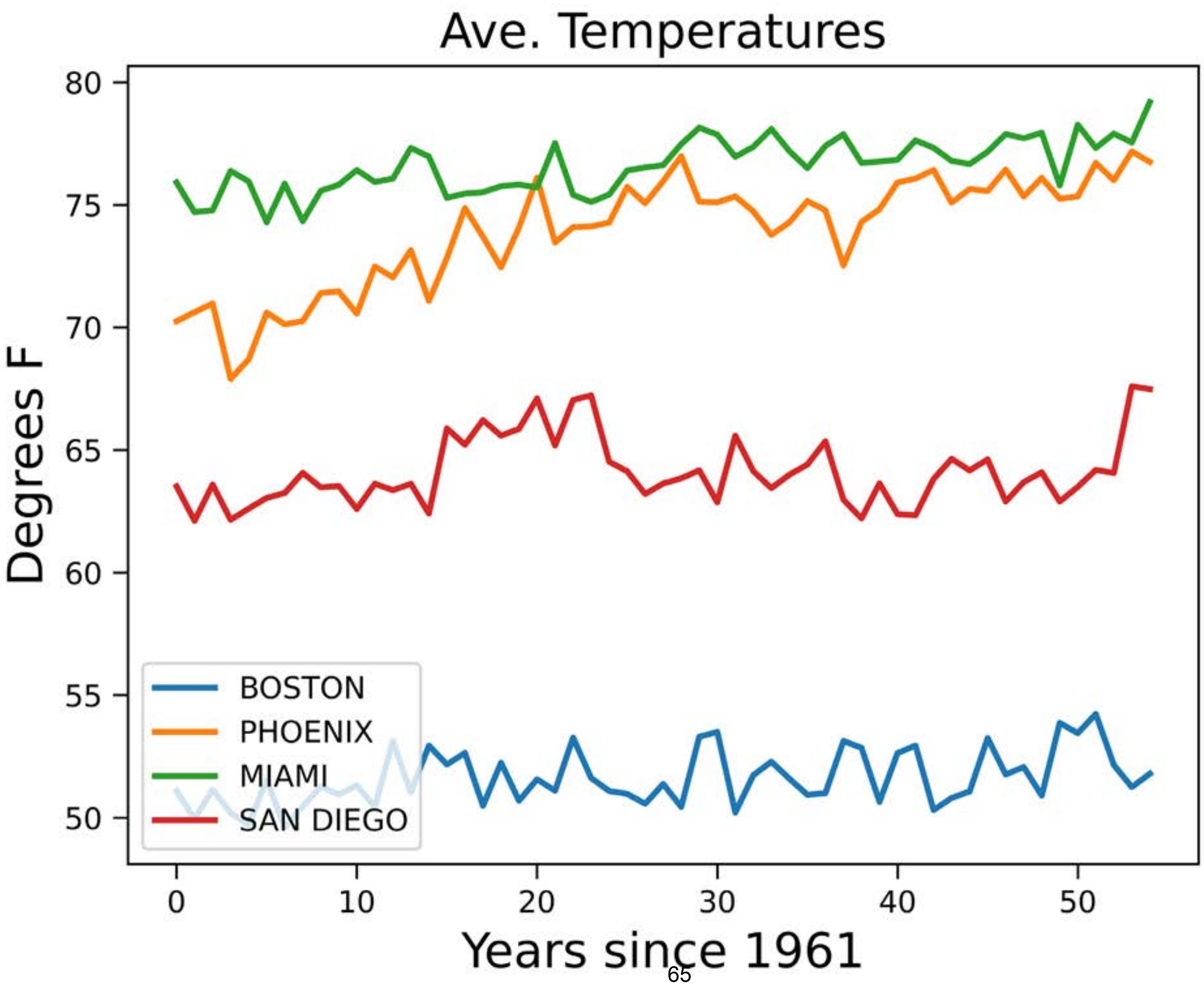
Get temp data for year

BUT MORE INTERESTING TO LOOK AT CHANGE OVER TIME

Pick some cities to plot 55 temps (avg temp over each year)

```
if True:
    plt.close()
    for c in ('BOSTON','PHOENIX', 'MIAMI', 'SAN DIEGO'):

        av, yr = getTempsByYearForCity(c)
        xPts = range(len(yr))
        plt.figure('Temps by City')
        plt.plot(xPts, av, label = c)
        plt.title('Ave. Temperatures')
        plt.xlabel('Years since 1961')
        plt.ylabel(('Degrees F'))
        plt.legend(loc = 'best')
```

BABY IT'S COLD OUTSIDE!



BUT WHAT IS VARIATION?
high, low, avg temps by year

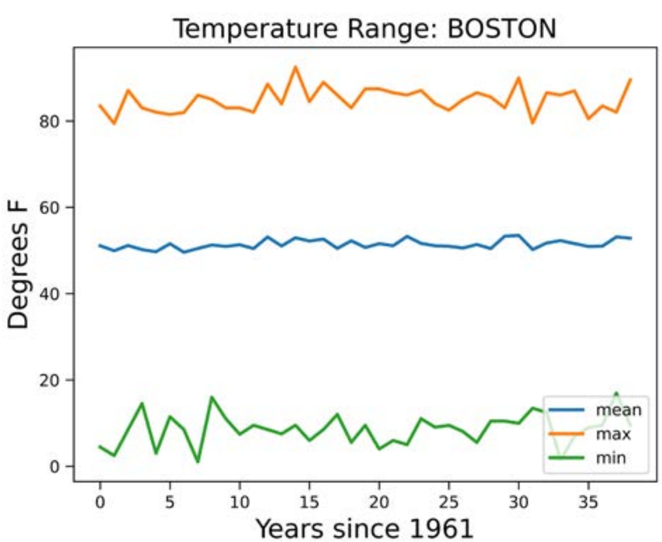
```
def getTempsForYearRange(tem, dat, y):
    yearly = []
    for i in range(len(tem)):
        if y == dat[i][:4]:
            yearly.append(tem[i])
    return sum(yearly)/len(yearly), max(yearly), min(yearly), y

def getTempsByYearForCityRange(city):
    temps, dates = getTempsForCity(city)
    averages = []
    maxes = []
    mins = []
    years = []
    for y in range(1961,2000):
        tem, mx, mn, y = getTempsForYearRange(temps, dates, str(y))
        averages.append(tem)
        maxes.append(mx)
        mins.append(mn)
        years.append(str(y))
    return averages, maxes, mins, years
```

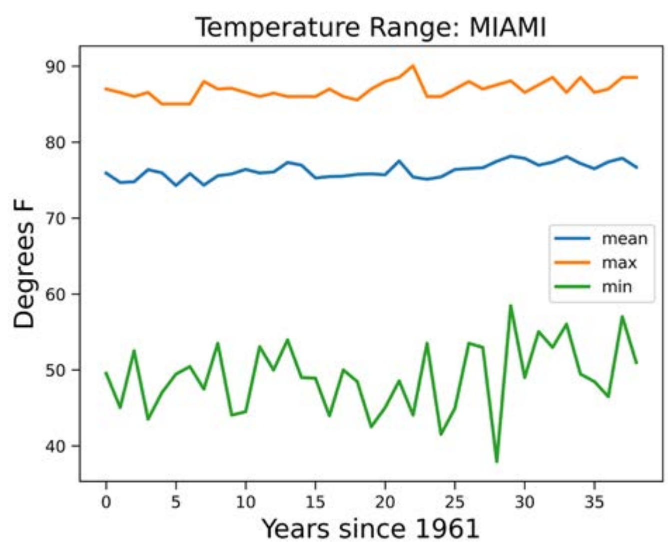
BUT WHAT IS VARIATION?
high, low, avg temps by year

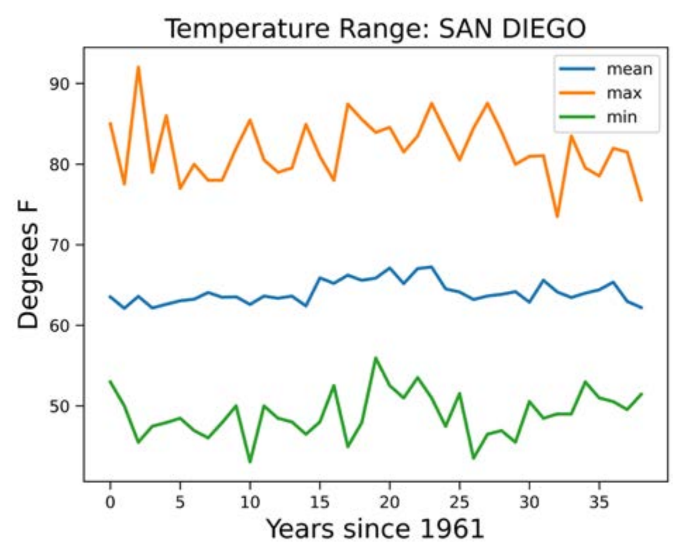
```
if True:
    plt.close()
    for c in ('BOSTON',):  # try for BOSTON, SAN DIEGO, MIAMI
        av, mx, mn, yr = getTempsByYearForCityRange(c)
        xPts = range(len(yr))
        plt.figure('Temps by City')
        plt.plot(xPts, av, label = 'mean')
        plt.plot(xPts, mx, label = 'max')
        plt.plot(xPts, mn, label = 'min')
        plt.title('Temperature Range: ' + c)
        plt.xlabel('Years since 1961')
        plt.ylabel(('Degrees F'))
        plt.legend(loc = 'best')
```

SOME CITY EXAMPLES

- Can see range for each city
- Not helpful for comparison between cities
  - Y axis for Boston is 0 to 80
  - Y axis for Miami is 40 to 90
  - Y axis for San Diego is 50 to 90







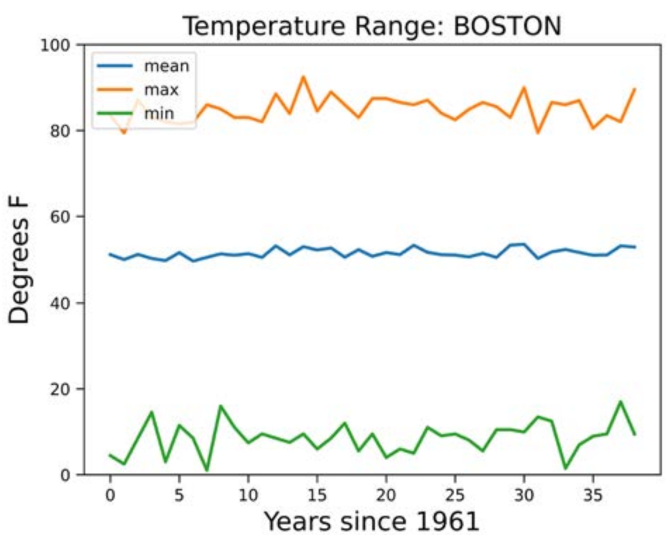
USE SAME Y RANGE FOR ALL PLOTS

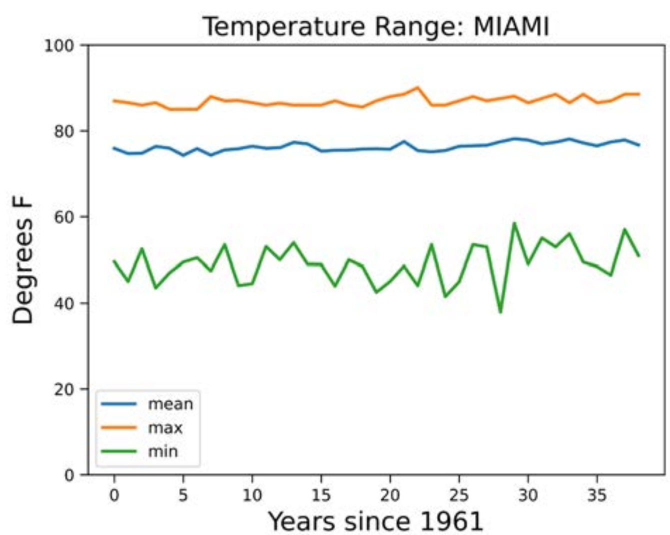
Fix the display range for y axis

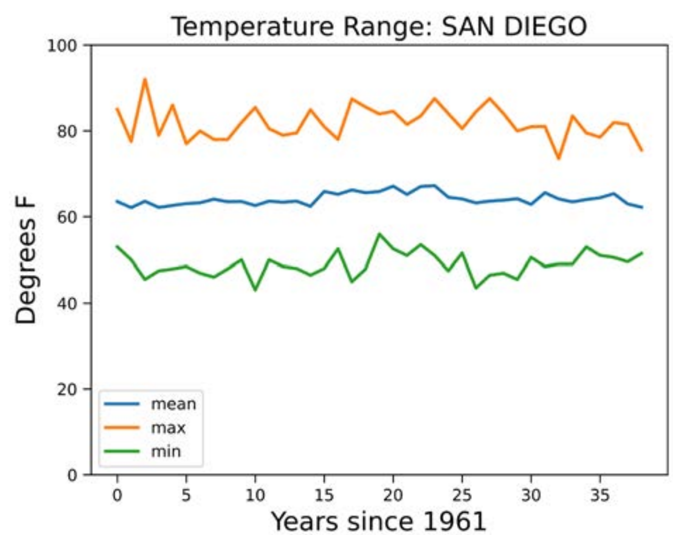
```
if True:
    plt.close()
    for c in ('MIAMI',):  # try for BOSTON, SAN DIEGO, MIAMI
        av, mx, mn, yr = getTempsByYearForCityRange(c)
        xPts = range(len(yr))
        plt.figure('Temps by City')
        plt.ylim(0, 100)
        plt.plot(xPts, av, label = 'mean')
        plt.plot(xPts, mx, label = 'max')
        plt.plot(xPts, mn, label = 'min')
        plt.title('Temperature Range: ' + c)
        plt.xlabel('Years since 1961')
        plt.ylabel(('Degrees F'))
        plt.legend(loc = 'best')
```

BETTER CITY COMPARISON

- One reason to plot is to visualize data
- Can see that range of variation is quite different for Boston, compared to Miami or San Diego
- Can also see that mean for Miami much closer to max than min. Different from Boston and San Diego







HOW MANY DAYS AT A TEMP in 1961?

Set up a list of 100 elements, making a histogram-like structure.

- Index 0 stores how many days had a temp of 0
- Index 1 stores how many days had a temp of 1

  …

- Index 99 stores how many days had a temp of 99.

```
def getDayDistributionForCity(city, year):
    # assume a range of temperatures from 0 to 100
    temps, dates = getTempsForCity(city)
    newTemps = []
    for i in range(len(dates)):
        if year == dates[i][:4]:
            newTemps.append(temps[i])
    ## want to map temperature to number of occurences
    d = [0]*100
    for t in newTemps:
        tRound = round(t)
        d[tRound] += 1
    return d
```

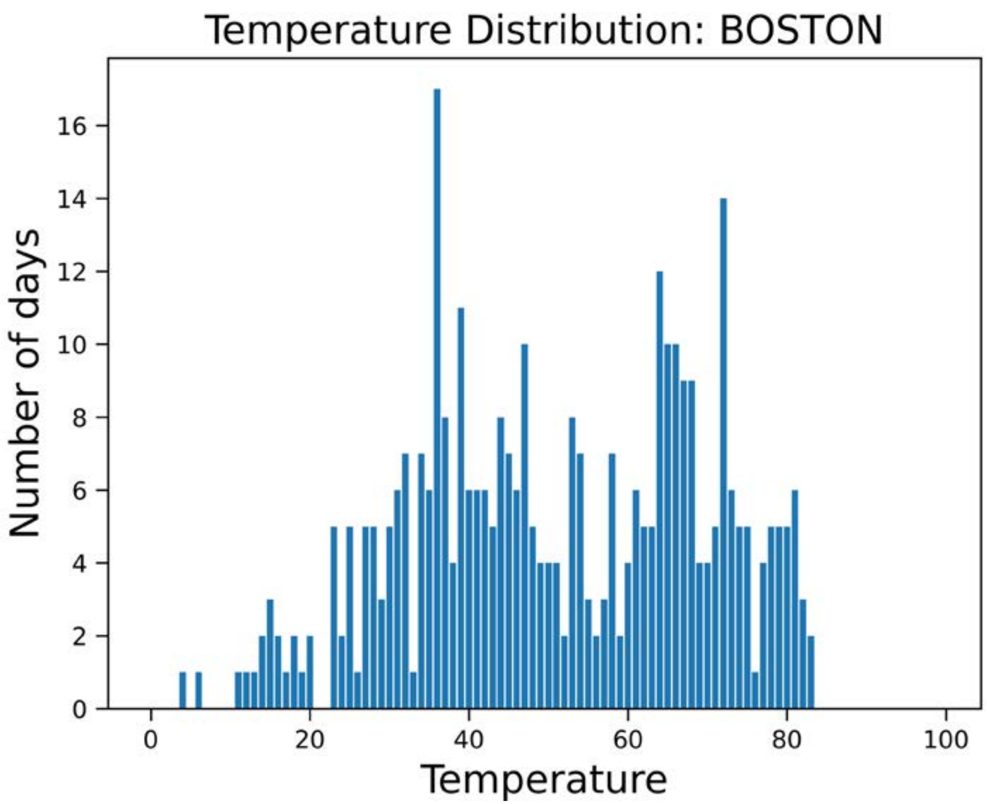
Create a list of temperatures for a specific year

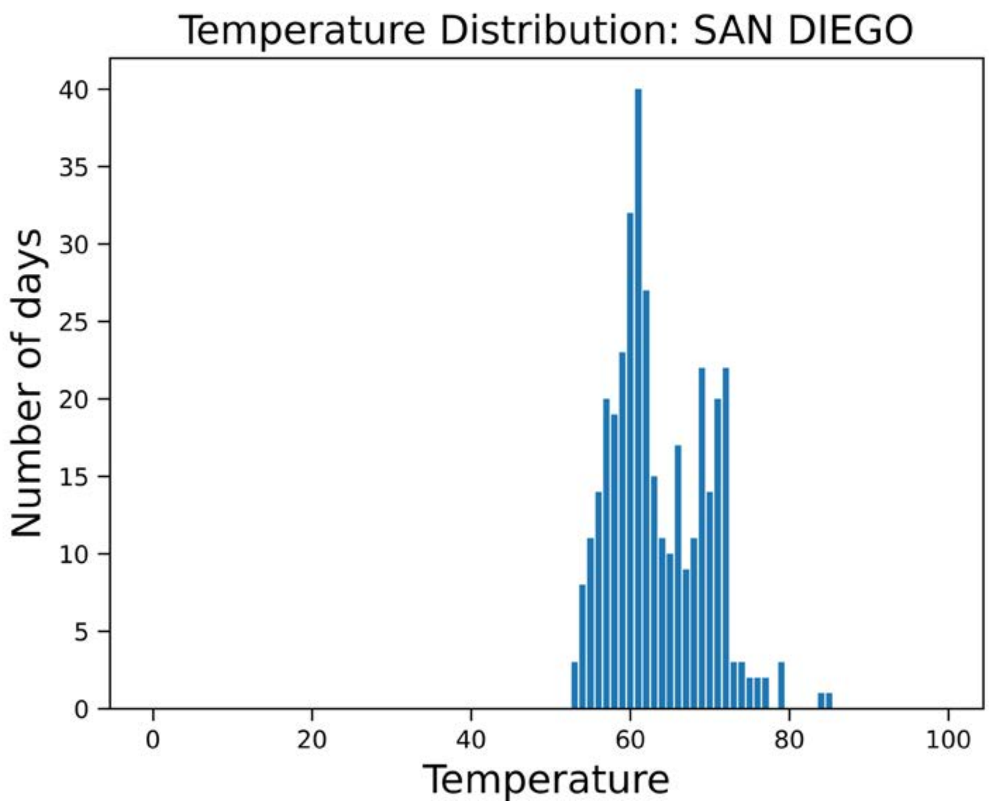
Count number of days of a particular year for which a specific temperature was the daily average

HOW MANY DAYS AT A TEMP IN 1961?

```python
if True:
    plt.close()
    for c in ('BOSTON',):  # try for BOSTON, SAN DIEGO, MIAMI
        ans = getDayDistributionForCity(c, '1961')
        temps = []
        for i in range(100):
            temps.append(i)
        plt.figure('Distribution of Temps by City')
        plt.bar(temps, ans)

        plt.title('Temperature Distribution: ' + c)
        plt.xlabel('Temperature')
        plt.ylabel(('Number of days'))
```

SAN DIEGO IS BORING?





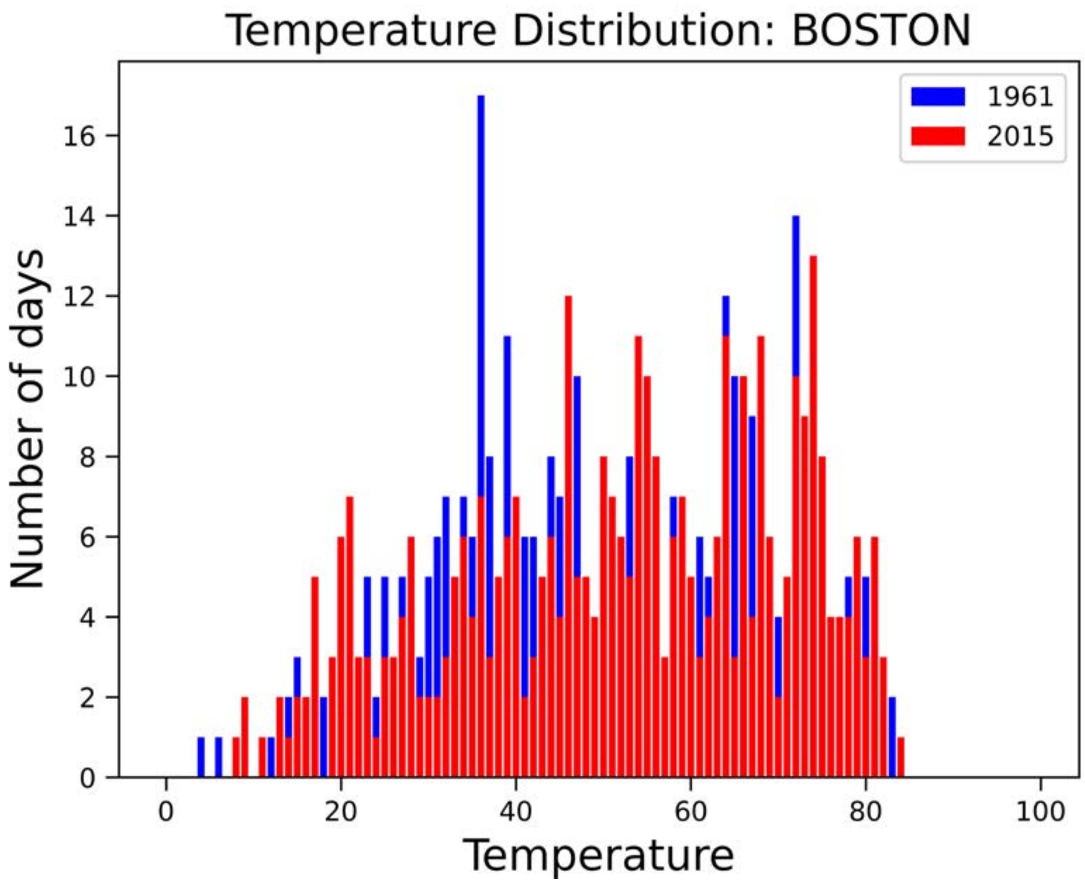
Could we fit a curve to parts of this data?
Uniform? Gaussian (aka bell)?

CHANGE OVER TIME?

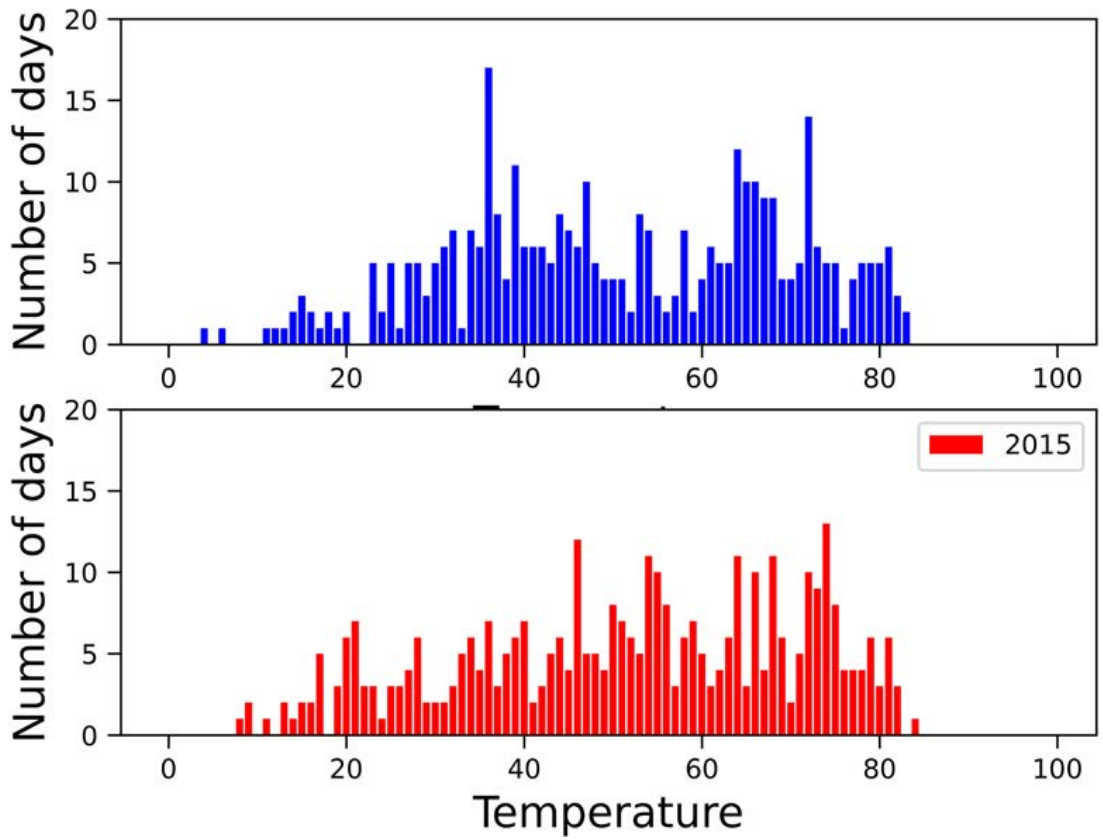
Plot two distributions, one for 1961 and one for 2015

```
if True:
    plt.close()
    for c in ('BOSTON',):  # try for BOSTON, SAN DIEGO
        plt.figure('Distribution of Temps by City')
        for y in ('1961', '2015'):
            ans = getDayDistributionForCity(c, y)
            temps = []
            for i in range(100):
                temps.append(i)
            if y == '1961':
                plt.bar(temps, ans, color = 'blue', label = y)
            else:
                plt.bar(temps, ans, color = 'red', label = y)

        plt.title('Temperature Distribution: ' + c)
        plt.xlabel('Temperature')
        plt.ylabel(('Number of days'))
        plt.legend(loc = 'best')
```

OVERLAY BAR CHARTS



OR CAN PLOT SEPARATELY



CAN CONTROL LOTS OF OTHER THINGS

- Size of
  - Markers
  - Lines
  - Title
  - Labels
  - x and y ticks
- Scales of both axes
- Subplots
- Text boxes
- Kind of plot
  - Scatter plots
  - Bar plots
  - Histograms
  - …

Scratched the surface today!

6.100L Introduction to Computer Science and Programming Using Python
Fall 2022

LIST ACCESS, HASHING, SIMULATIONS, & WRAP-UP!

(download slides and .py files to follow along)

6.100L Lecture 26

Ana Bell

TODAY

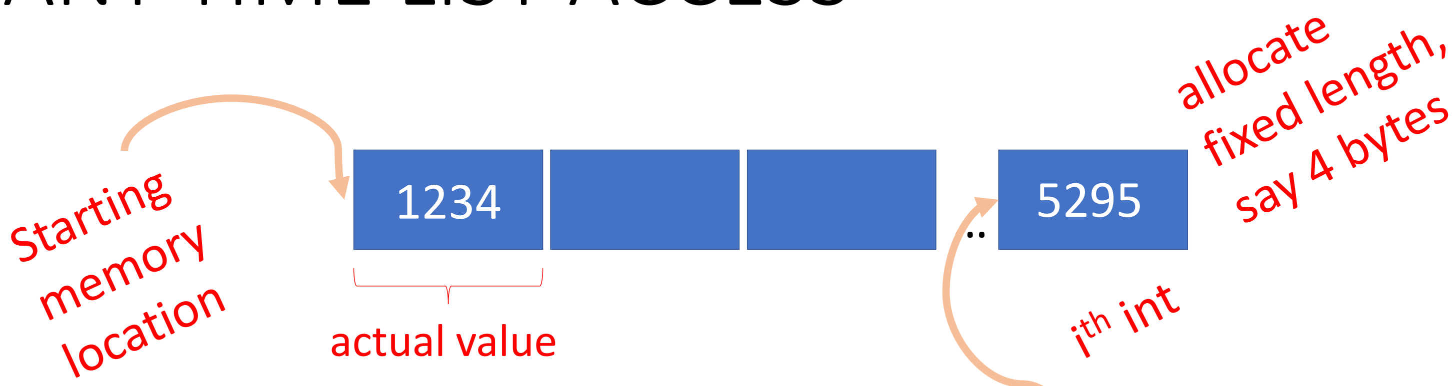
- A bit about lists
- Hashing
- Simulations

LISTS

COMPLEXITY OF SOME PYTHON OPERATIONS

- Lists: `n` is `len(L)`
  - access θ(1)
  - store θ(1)
  - length θ(1)
  - append θ(1)
  - **== θ(n)**
  - **delete θ(n)**
  - **copy θ(n)**
  - **reverse θ(n)**
  - **iteration θ(n)**
  - **`in` list θ(n)**

CONSTANT TIME LIST ACCESS



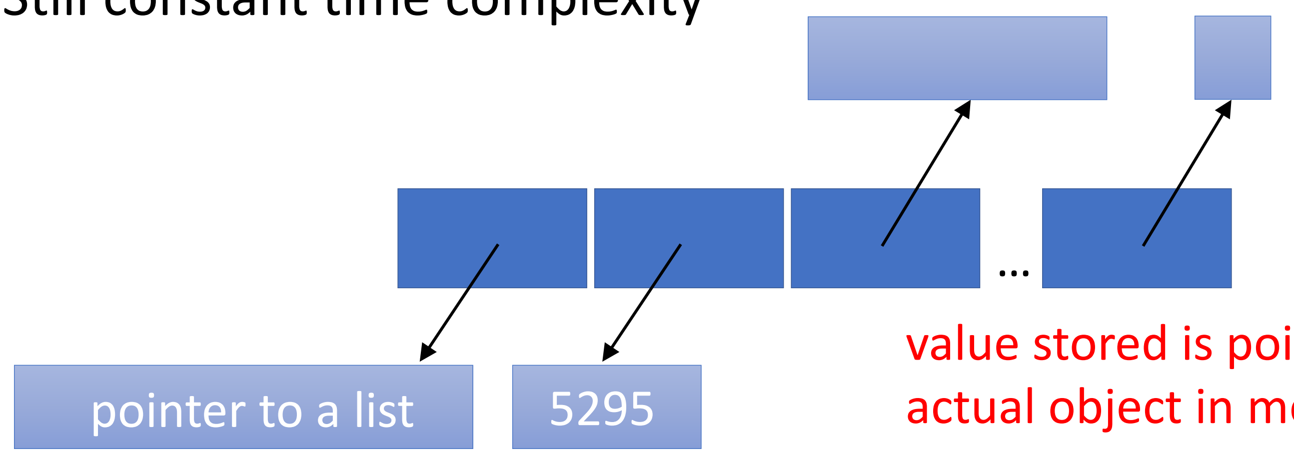
This location is
32*i + start location

- If list is all `ints`, list of length L
  - Set aside 4*len(L) bytes
  - Store values directly
  - Consecutive set of memory locations
- List name points to first memory location
- To access i^{th} element
  - **Add 32*i to first location**
  - Access that location in memory
  - **Constant** time complexity

since entries are 4*8=32 bits long

CONSTANT TIME LIST ACCESS

- If list is **heterogeneous**
  - Can't store values directly (don't all fit in 32 bits)
  - Use **indirection** to reference other objects
  - **Store pointers** to values (not value itself)
  - Still use consecutive set of memory locations
  - Still set aside 4*len(L) bytes
  - Still add 32*i to first location and +1 to access that location in memory
  - Still constant time complexity



value stored is pointer to actual object in memory

still allocate fixed length units

NAÏVE IMPLEMENTATION OF dict

- Just use a list of pairs: key, value

```
[['Ana', True], ['John', False], ['Eric', False], ['Sam', False]]
```

- What is time **complexity to index** into this naïve dictionary?
  - We don't know the order of entries
  - Have to do **linear search** to find entry

COMPLEXITY OF SOME PYTHON OPERATIONS

- Lists: `n` is `len(L)`
  - access θ(1)
  - store θ(1)
  - length θ(1)
  - append θ(1)
  - **== θ(n)**
  - **delete θ(n)**
  - **copy θ(n)**
  - **reverse θ(n)**
  - **iteration θ(n)**
  - **in list θ(n)**

- Dictionaries: `n` is `len(d)`
- worst case (very rare)
  - **length θ(n)**
  - **access θ(n)**
  - **store θ(n)**
  - **delete θ(n)**
  - **iteration θ(n)**
- average case
  - access θ(1)
  - store θ(1)
  - delete θ(1)
  - in θ(1)
  - **iteration θ(n)**

Why?

HASHING

DICTIONARY IMPLEMENTATION

- Uses a **hash table**
- How it does it
  - Convert **key to an integer** – use a **hash function**
  - Use that integer as the index into a list
    - This is constant time
  - Find value associated with key
    - This is constant time
- Dictionary lookup is **constant time complexity**
  - If hash function is fast enough
  - If indexing into list is constant

QUERYING THE HASH FUNCTION

- Just to reveal what's under the hood, a function `hash()`

```
In [9]: hash(123)
Out[9]: 123

In [10]: hash("6.100L is awesome")
Out[10]: 8708784260240907980

In [11]: hash((1,2,3))
Out[11]: 529344067295497451

In [12]: hash([1,2,3])
Traceback (most recent call last):

  File "<ipython-input-12-35e31e935e9e>",
line 1, in <module>
    hash([1,2,3])

TypeError: unhashable type: 'list'
```

May vary because Python adds randomness to thwart attacks

Why do this? Because hashing is also used to encrypt data for safe storage and retrieval.

HASH TABLE

- **How big** should a hash table be?
- To avoid many keys hashing to the same value, have each key hash to a separate value
- If hashing strings:
  - Represent **each character** with binary code
  - **Concatenate bits together**, and **convert to an integer**

NAMES TO INDICES

- E.g., `'Ana Bell'`
  `= 01000001 01101110 01100001 00100000 01000010 01100101 01101100 01101100`
  `= 4,714,812,651,084,278,892`
- **Advantage**: unique names mapped to **unique indices**
- **Disadvantage**: VERY **space inefficient**
- Consider a table containing MIT's ~4,000 undergraduates
  - Assume longest name is 20 characters
  - Each character 8 bits, so 160 bits per name
  - How many entries will table have?

2^{160}

1,461,501,637,330,902,918,203,684,832,716,283,019,655,932,542,976

A BETTER IDEA: ALLOW COLLISIONS

Hash function:

1) Sum the letters
2) Take mod 16 (to fit in a hash table with 16 entries)

1 + 14 + 1 = 16
16%16 = 0

A n a	C

5 + 18 + 9 + 3 = 35
35%16 = 3

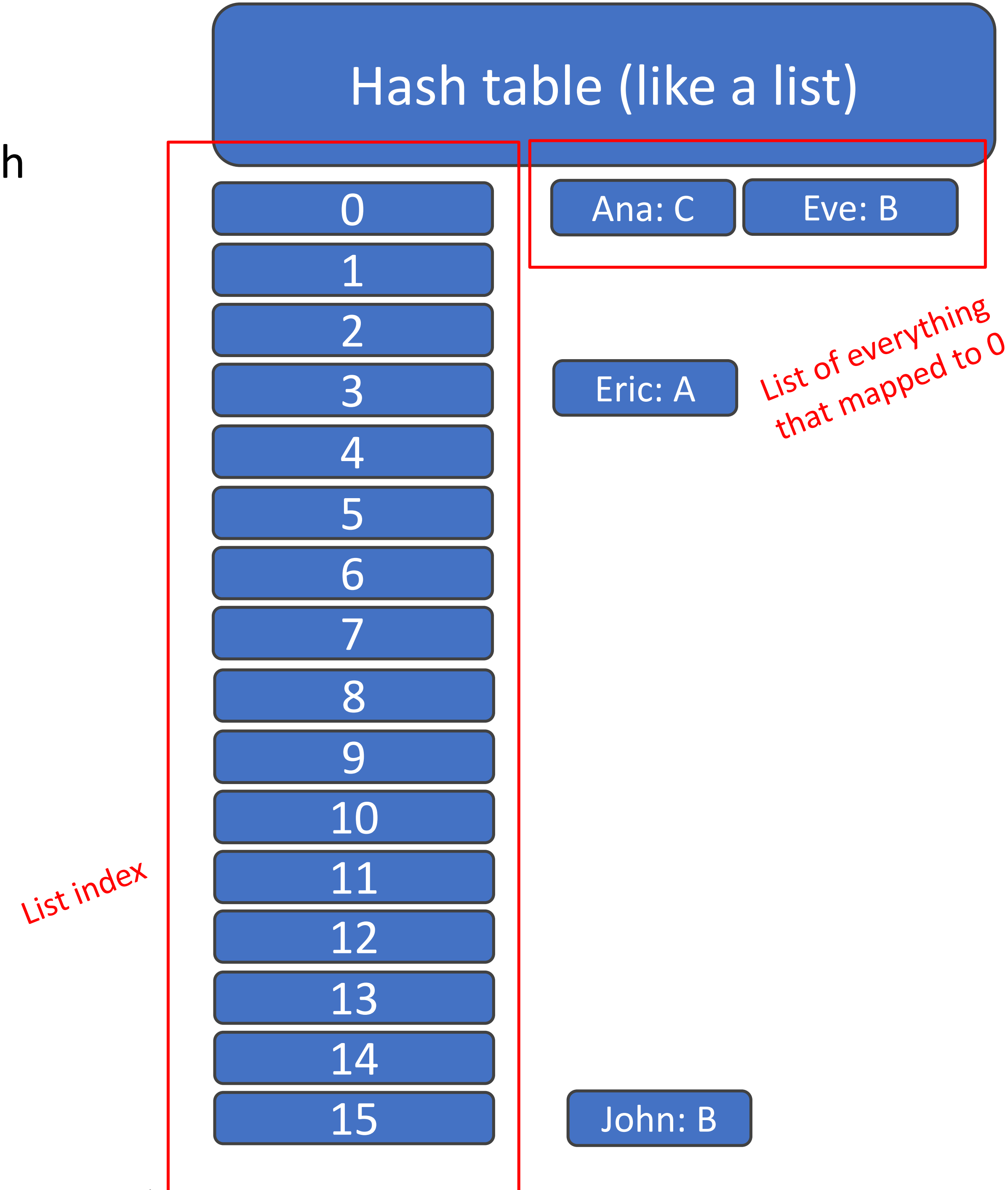
E r i c	A

10 + 15 + 8 + 14 = 47
47%16 = 15

J o h n	B

5 + 22 + 5 = 32
32%16 = 0

Eve	B



PROPERTIES OF A GOOD HASH FUNCTION

- Maps domain of interest to integers between 0 and size of hash table
- The hash value is **fully determined by value** being hashed (nothing random)
- The hash function uses the entire input to be hashed
  - Fewer collisions
- **Distribution of values is uniform**, i.e., equally likely to land on any entry in hash table
- Side Reminder: keys in a dictionary must be **hashable**
  - aka immutable
  - They always hash to the same value
  - What happens if they are not hashable?

Hash function:

1) Sum the letters
2) Take mod 16 (to fit in a memory block with 16 entries)

1 + 14 + 1 = 16
16%16 = 0

A n a	C

5 + 18 + 9 + 3 = 35
35%16 = 3

E r i c	A

10 + 15 + 8 + 14 = 47
47%16 = 15

J o h n	B

5 + 22 + 5 = 32
32%16 = 0

Eve	B

11 + 1 + 20 + 5 = 37
37%16 = 5

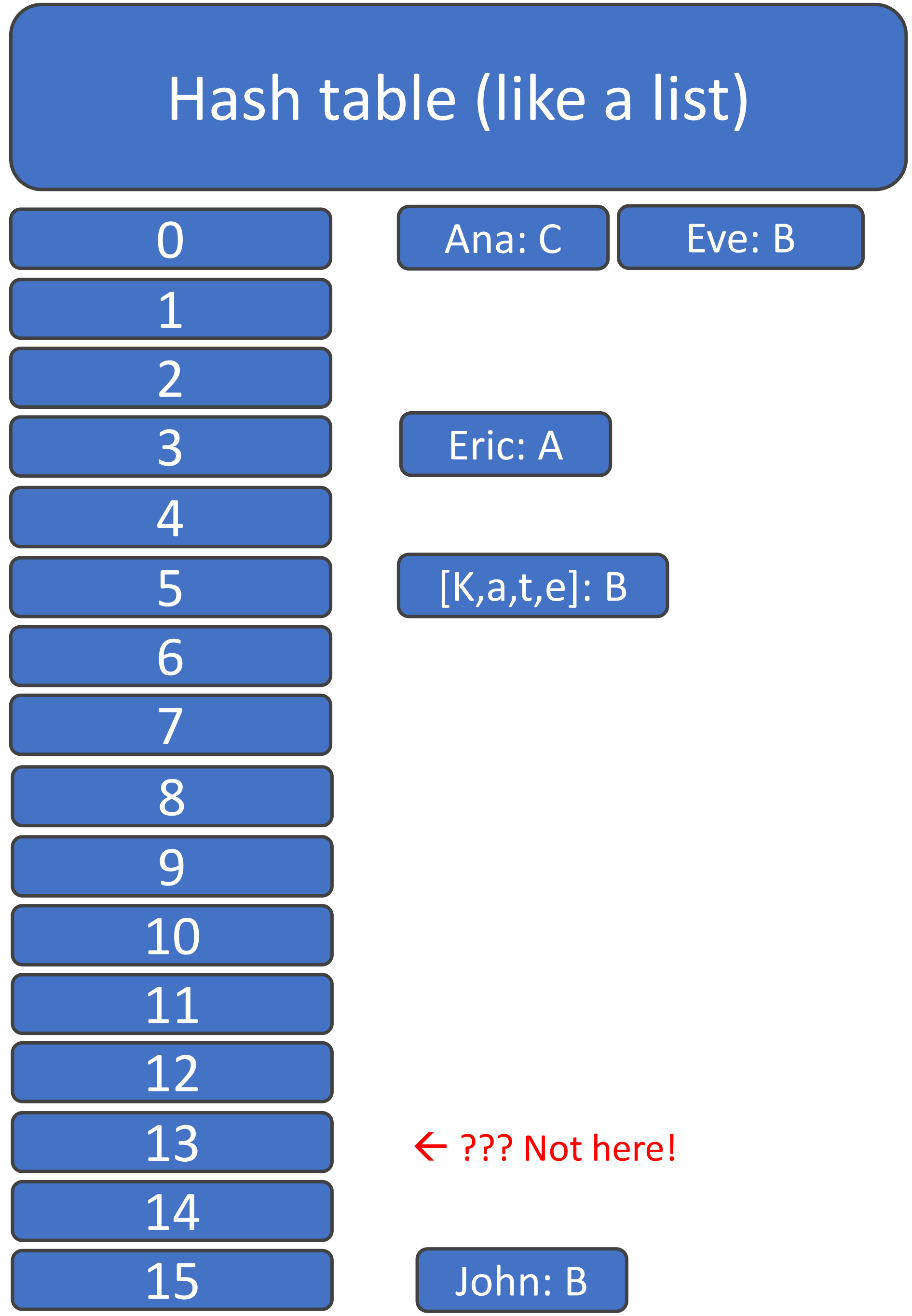
[K, a, t, e]	B

Hash table (like a list)

0	Ana: C	Eve: B
1		
2		
3	Eric: A	
4		
5	[K,a,t,e]: B	
6		
7		
8		
9		
10		
11		
12		
13		
14		
15	John: B	

Hash function:
1) Sum the letters
2) Take mod 16 (to fit in a memory block with 16 entries)

Kate changes her name to Cate. Same person, different name. Look up her grade?



3 + 1 + 20 + 5 = 29
29%16 = 13

[C, a, t, e]

COMPLEXITY OF SOME PYTHON OPERATIONS

- Dictionaries: `n` is `len(d)`
- worst case (very rare)
  - **length** **θ(n)**
  - **access** **θ(n)**
  - **store** **θ(n)**
  - **delete** **θ(n)**
  - **iteration** **θ(n)**

If all keys hash to the same index

- average case
  - access θ(1)
  - store θ(1)
  - delete θ(1)
  - in θ(1)
  - **iteration** **θ(n)**

Hash table is large relative to number of keys

Hash function good enough

SIMULATIONS

TOPIC USEFUL FOR MANY DOMAINS

- **Computationally** describe the world using **randomness**
- One very important topic relevant to many fields of study
  - Risk modeling and analysis
  - Reduce complex models
- Idea:
  - **Observe an event** and want to calculate something about it
  - Using computation, **design an experiment** of that event
  - **Repeat the experiment** K many times (make a simulation)
  - **Keep track** of the outcome of your event
  - After K repetitions, **report the value of interest**

ROLLING A DICE

- Observe an event and want to calculate something about it
  - Roll a dice, what's the **prob to get a ::**? How about a .?
- Using computation, design an experiment of that event
  - Make a **list** representing die faces and **randomly choose one**
  - `random.choice(['.',':',':.','::','::.',':::'])`
- Repeat the experiment K many times (simulate it!)
  - Randomly choose a die face from a list **repeatedly**, 10000 times
  - How? Wrap the simulation in a **loop**!

```
for i in range(10000):
 roll=random.choice(['.',':',':.','::','::.',':::'])
```

- Keep track of the outcome of your event
  - **Count** how many times out of 10000 the roll equaled ::
- After K repetitions, report the value of interest
  - **Divide** the count by 10000

THE SIMULATION CODE

```
def prob_dice(side):
    dice = ['.',':',':.','::','::.',':::']
    Nsims = 10000
    count = 0
    for i in range(Nsims):
        roll = random.choice(dice)
        if roll == side:
            count += 1
    print(count/Nsims)
```

Repeat experiment

Choose random dice face

Count successes

```
prob_dice('.')
prob_dice('::')
```

0.1677

0.1602

THAT'S AN EASY SIMULATION

- We can compute the probability of a die roll mathematically
- **Why bother** with the code?
- Because we can answer **variations** of that original question and we can ask **harder** questions!
  - Small tweaks in code
  - Easy to change the code
  - Fast to run

NEW QUESTION
NOT AS EASY MATHEMATICALLY

- Observe an event and want to calculate something about it
  - Roll a dice 7 times, **what's the prob to get a :: at least 3 times out of 7 rolls**?
- Using computation, design an experiment of that event
  - Make a list representing die faces and **randomly choose one 7 times in a row**
  - **Face counter increments** when you choose :: (keep track of this number)
- Repeat the experiment K many times (simulate it!)
  - **Repeat** the prev step 10000 times.
  - How? Wrap the simulation in a **loop**!
- Keep track of the outcome of your event
  - **Count** how many times out of 10000 the :: face counter >= 3
- After K repetitions, report the value of interest
  - **Divide** the outcome count by 10000

EASY TWEAK TO EXISTING CODE

Generalize fcn

```
def prob_dice_atleast(Nrolls, n_at_least):
    dice = ['.',':',':.','::','::.',':::']
    Nsims = 10000
    how_many_matched = []
    for i in range(Nsims):
        matched = 0
        for i in range(Nrolls):
            roll = random.choice(dice)
            if roll == '::':
                matched += 1
        how_many_matched.append(matched)

    count = 0
    for i in how_many_matched:
        if i >= n_at_least:
            count += 1
    print(count/len(how_many_matched))
```

Roll 7 times and keep track, in a list, how many :: came up

How many times :: came up >=3 times out of 10000

```
prob_dice_atleast(7, 3)
prob_dice_atleast(1, 1)
```

0.0955

0.16

REAL WORLD QUESTION
VERY COMMON EXAMPLE OF HOW USEFUL SIMULATIONS CAN BE

- Water runs through a faucet somewhere between 1 gallons per minute and 3 gallons per minute
- What's the **time it takes to fill a 600 gallon pool**?
    - Intuition?
    - It's not 300 minutes (600/2)
    - It's not 400 minutes (600/1 + 600/3)/2
- In code:
    - Grab a bunch of random values between 1 and 3
    - Simulate the time it takes to fill a 600 gallon pool with each randomly chose value
    - Print the average time it takes to fill the pool over all these randomly chosen values

```
def fill_pool(size):
    flow_rate = []
    fill_time = []
    Npoints = 10000
    for i in range(Npoints):
        r = 1+2*random.random()
        flow_rate.append(r)
        fill_time.append(size/r)

    print('avg flow_rate:', sum(flow_rate)/len(flow_rate))
    print('avg fill_time', sum(fill_time)/len(fill_time))
    plt.figure()
    plt.scatter(range(Npoints),flow_rate,s=1)
    plt.figure()
    plt.scatter(range(Npoints),fill_time,s=1)

fill_pool(600)
```

How many random values to get

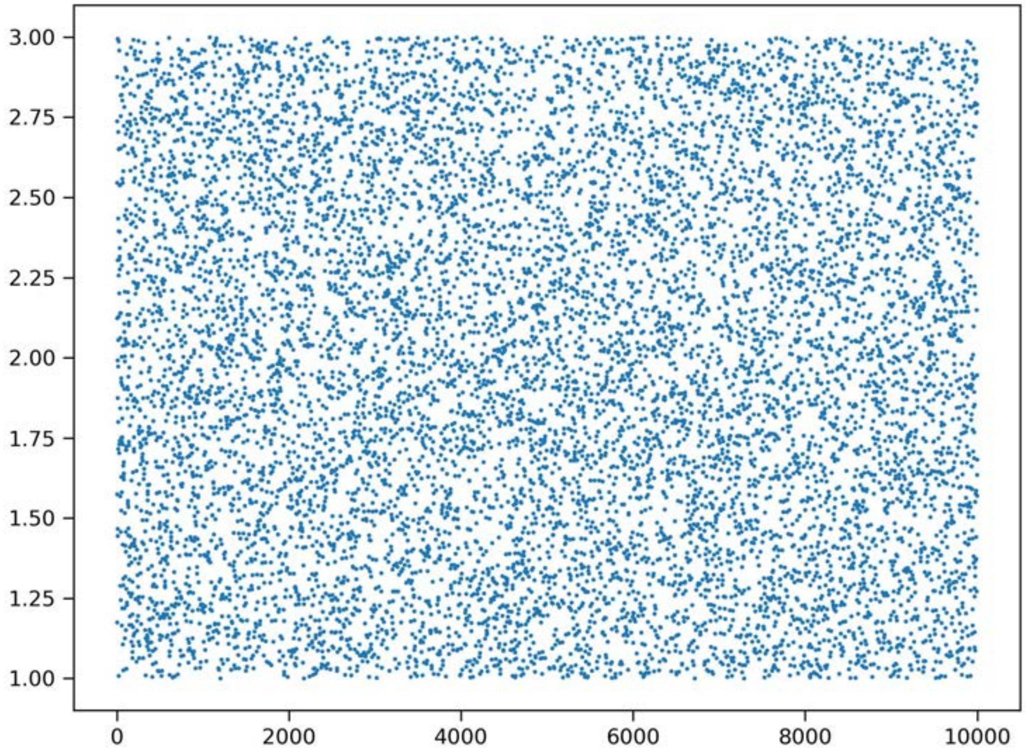
Number between 1 and 3

Use the random value to determine how long it takes to fill up

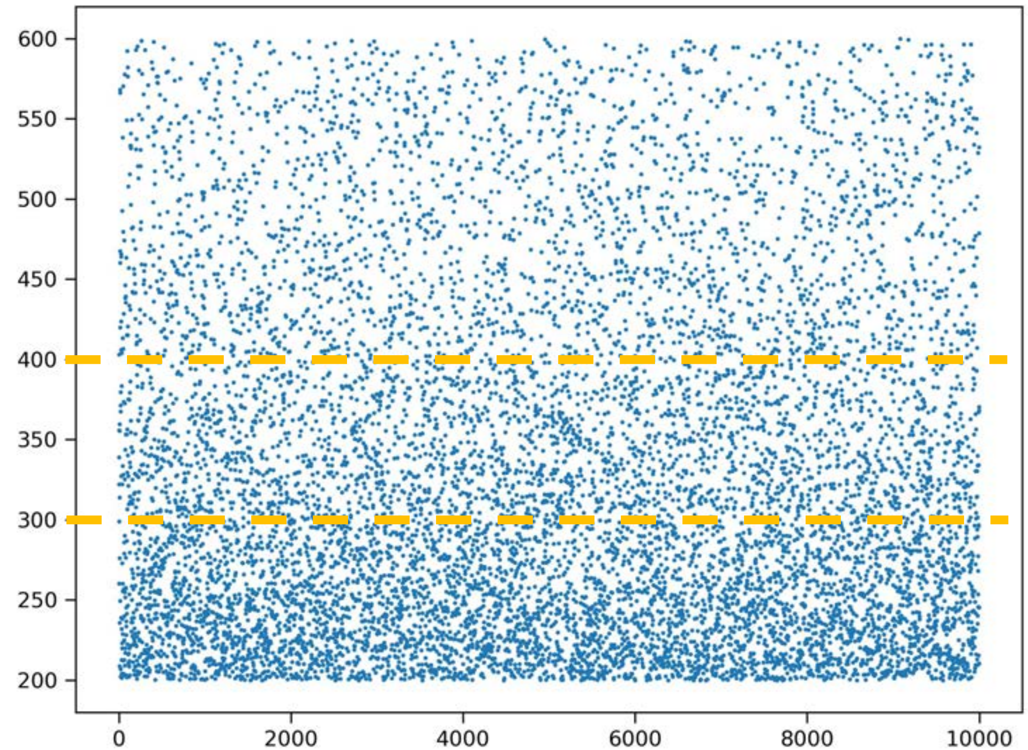
Average over Npoints

PLOTTING RANDOM FILL RATES AND CORRESPONDING TIME IT TAKES TO FILL

Random values for **fill rate**

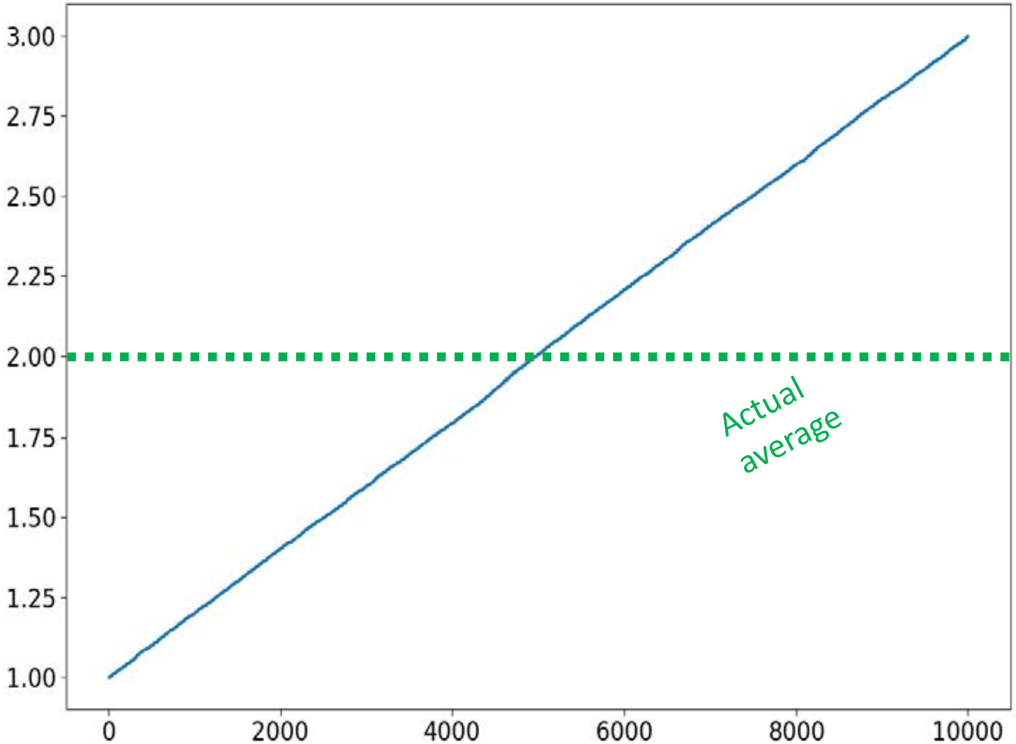


Time to fill using formula pool_size/rate

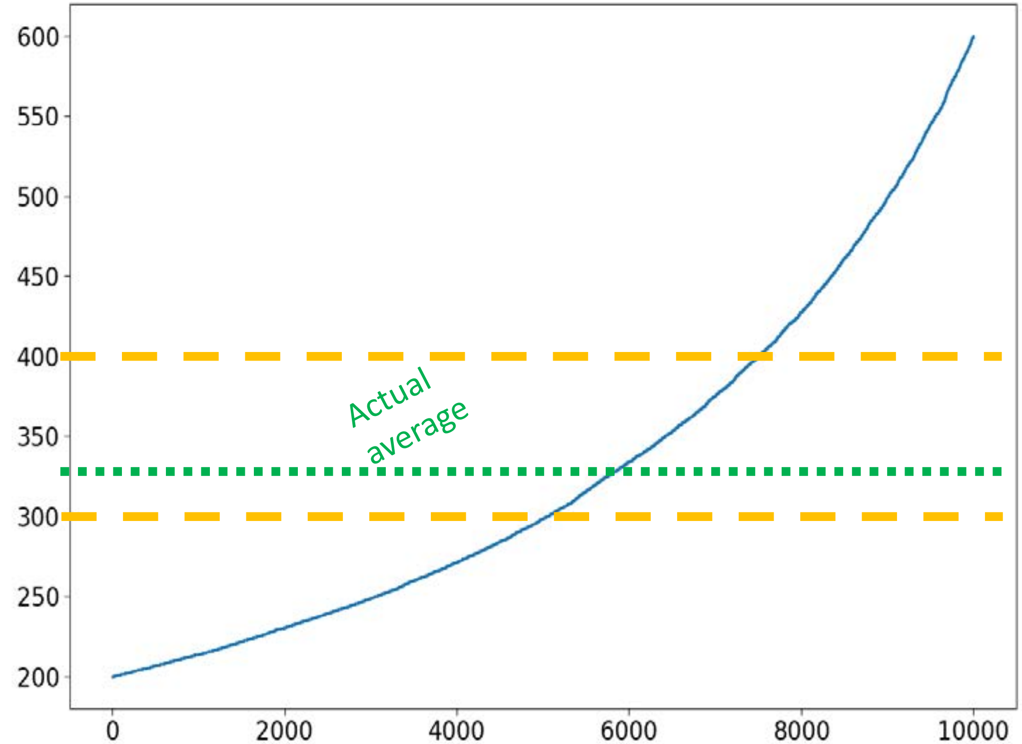


PLOTTING RANDOM FILL RATES AND CORRESPONDING TIME IT TAKES TO FILL

Random values for **fill rate (sorted)**



Time to fill (sorted) using formula pool_size/rate



RESULTS

- avg flow_rate: 1.992586945871106 **approx. 2 gal/min** (avg random values between 1 and 3)
- avg fill_time: 330.6879477596955 **approx. 331 min** (not what we expected!)

- Not 300 and not 400
- There is an **inverse relationship** for fill time vs fill rate
  - Mathematically you'd have to do an **integral**
  - Computationally you just write a **few lines of code**!

WRAP-UP of 6.100L

THANK YOU FOR BEING IN THIS CLASS!

WHAT DID YOU LEARN?

- Python syntax
- Flow of control
  - Loops, branching, exceptions
- Data structures
  - Tuples, lists, dictionaries
- Organization, decomposition, abstraction
  - Functions
  - Classes
- Algorithms
  - Binary/bisection
- Computational complexity
  - Big Theta notation
  - Searching and sorting

YOUR EXPERIENCE

- Were you a "natural"?
- Did you join the class late?
- Did you work hard?

- Look back at the first pset
  it will seem so easy!
- You **learned a LOT** no matter what!

WHAT’S NEXT

- **6.100B** overview of interesting topics in CS and data science (Python)
  - Optimization problems
  - Simulations
  - Experimental data
  - Machine learning

WHAT'S NEXT

- **6.101** fundamentals of programming (Python)
  - Implementing efficient algorithms
  - Debugging

WHAT'S NEXT

- **6.102** software construction (TypeScript)
  - Writing code that is safe from bugs, easy to understand, ready for change

WHAT'S NEXT

- Other classes
  (ML, algorithms, etc.)

IT’S EASY TO FORGET WITHOUT PRACTICE! HAPPY CODING!